WebSphere® Adapters

IBM

Version 6 Release 2

WebSphere Adapter Toolkit User Guide
Version 6 Release 2

WebSphere® Adapters

IBM

Version 6 Release 2

WebSphere Adapter Toolkit User Guide
Version 6 Release 2

Note

Before using this information and the product it supports, read the information in “Notices” on page 211.

December 2008

This edition applies to version 6, release 2, modification 0 of and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, email mailto://doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2006, 2008. All rights reserved. US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

© Copyright International Business Machines Corporation 2006, 2008.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

WebSphere Adapter Toolkit 1
IBM WebSphere Adapter Toolkit technology overviews 2
IBM WebSphere Adapters 2
Architectural overview 4
How metadata is used at build time and run time 7
Using Enterprise Metadata Discovery to build services 7
IBM WebSphere Adapter Toolkit overview 8
New Connector Project wizard overview 9
Resource Adapter Deployment Descriptor Editor overview 10
Adapter Foundation Classes overview 10
IBM WebSphere Adapter Toolkit tasks 11
IBM WebSphere Adapter Toolkit installation requirements 11
Samples overview 12
Running the Twine Ball sample using WebSphere Integration Developer 13
Running the Twine Ball sample using Rational Application Developer. 26
Troubleshooting the samples. 28
Using the New Connector Project wizard 28
Launching the New Connector Project wizard . . 29
Specify project properties. 31
Specify project facets 32
Specify connector project module settings . . . 32
Specify resource adapter properties 33
Specify generation options 34
Generating an IBM WebSphere Resource Adapter 35
Generating a JCA resource adapter 49
Generated code and deployment descriptor. . . 55
Using the Resource Adapter Deployment Descriptor editor 56
Displaying the deployment descriptor 56
Modifying deployment descriptor properties . . 66
Editing deployment descriptor source 67
Implementing code from the IBM WebSphere Adapter Toolkit 68
Foundation Classes implementation overview . . 68
Data model 69
Inbound event notification 79
Inbound callback event notification 89
Outbound support 97
Data and metadata 115
Enterprise Metadata Discovery general interfaces and implementation for application adapters 123
Enterprise Metadata Discovery interfaces and implementation for technology adapters . . . 157
Structured record implementation 162
Data binding implementation 166
Bidirectional language support 167
Problem determination 168
Validating the code 193
Testing enterprise metadata discovery (EMD) of the adapter 193
Testing the adapter in unmanaged mode . . . 194
Testing the adapter in managed mode 197
Validating code with Rational Application Developer and WebSphere Application Server . 203
Creating and exporting a resource adapter . . . 206
Reference. 208
Terminology. 208

Notices 211
Programming interface information 213
Trademarks and service marks 213

Index 215

© Copyright IBM Corp. 2006, 2008

WebSphere Adapter Toolkit

The IBM® WebSphere® Adapter Toolkit provides the development tools, libraries and sample code to assist you in creating JCA resource adapters.

With the toolkit you may create either of the following:

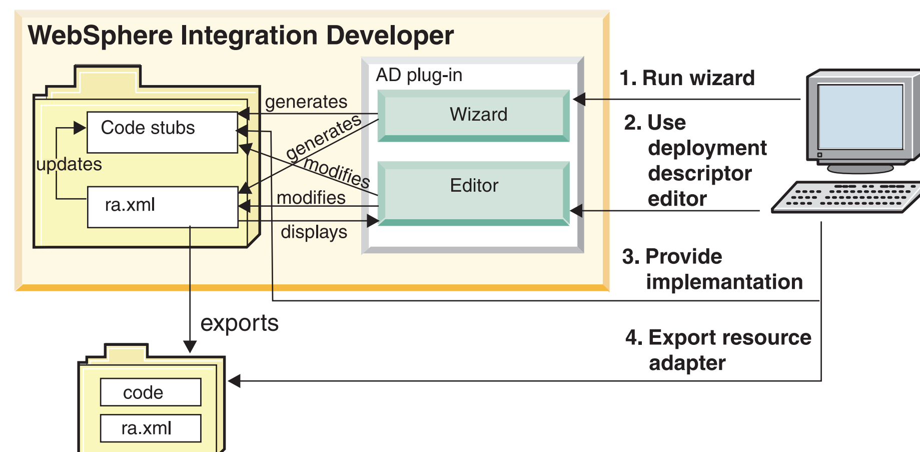
- **A resource adapter based on the interfaces defined by the JCA Resource Adapter 1.5 specification**. Choose this path if your goal is developing a resource adapter that can run either unmanaged or managed within any JCA 1.5 compliant container.
- **A resource adapter that extends the WebSphere Adapter Foundation Classes library**. Choose this path if your goal is to create a resource adapter implementation that can run in a managed server runtime environment like WebSphere Process Server and that exhibits the common functionality and extended qualities of service offered specifically by WebSphere Adapters. WebSphere Adapters are based on the JCA 1.5 specification. These adapters are supported on multiple runtime environments and brokers.

Implementing a WebSphere adapter allows you to take advantage of the quality of services offered in the WebSphere Adapter foundation classes (AFC) and the flexibility of being able to run the adapter on runtime environments other than WebSphere Process Server.

In either case, the toolkit provides a project creation wizard that generates the code that you then implement. In addition, the toolkit provides a specialized editor that facilitates the task of creating and configuring a resource adapter deployment descriptor.

This document focuses primarily on development of resource adapters and artifacts that extend the WebSphere Adapter Foundation Class library.

The following figure illustrates the process of developing a JCA resource adapter using the WebSphere Adapter Toolkit.



Using the WebSphere Adapter Toolkit

The development process using the IBM WebSphere Adapter Toolkit includes the following as shown in the illustration:

1. Run the New JCA Resource Adapter Project wizard.

   The wizard generates a resource adapter deployment descriptor and code. The code can include sequence of calls, log and trace messages and comments.
2. Use the Resource Adapter Deployment Descriptor Editor to configure your deployment descriptor.
3. Implement the code to correctly interface with your enterprise information system (EIS).
4. Export the resource adapter as a resource adapter archive (RAR) or enterprise application archive (EAR) file.

The purpose of this documentation

Adapter development requires a great deal of software engineering, which varies from customer to customer. The process of integrating adapter functionality into your business processes will require you to design, build and test the solution that utilizes the adapter. The purpose of the WebSphere Adapter Toolkit documentation is to lay out the requirements of the architecture and provide guidance on how and when to implement the various facets of it, so that you can apply your engineering discipline to the adapter-specific requirements, capabilities and architecture.

IBM WebSphere Adapter Toolkit technology overviews

The IBM WebSphere Adapter Toolkit helps developers implement the Adapter Foundation Classes, which establish a WebSphere Adapter standard for building resource adapters that conform to the Java 2 Connector Architecture (JCA) 1.5 specification.

IBM WebSphere Adapters

An IBM WebSphere Adapter implements the Java 2 Enterprise Edition (J2EE) Connector architecture (JCA), version 1.5. Also known as resource adapters or JCA adapters, WebSphere Adapters enable managed, bidirectional connectivity between enterprise information systems (EIS) and J2EE components supported by multiple server runtime environments, including WebSphere Process Server and WebSphere Application Server.

IBM WebSphere adapters support outbound request processing and inbound event processing.

Note: IBM WebSphere adapters support outbound and inbound processing for WebSphere Process Server and *outbound processing only* for WebSphere Application Server.

Outbound processing refers to a process in which request data flows from a client application to the EIS. In an outbound processing scenario, the adapter acts as the connector between the application component and the EIS. The adapter provides a set of standard operations, which process either after-image or delta style business objects. An outbound request can read data from or write data to the EIS.

Inbound event processing refers to a process that is initiated by an event on the EIS. In inbound event processing the adapter converts events generated from the EIS into business objects and sends the business object to the client application.

Inbound event notification complements outbound request processing, enabling adapters to provide bidirectional communication between business processes and EIS applications.

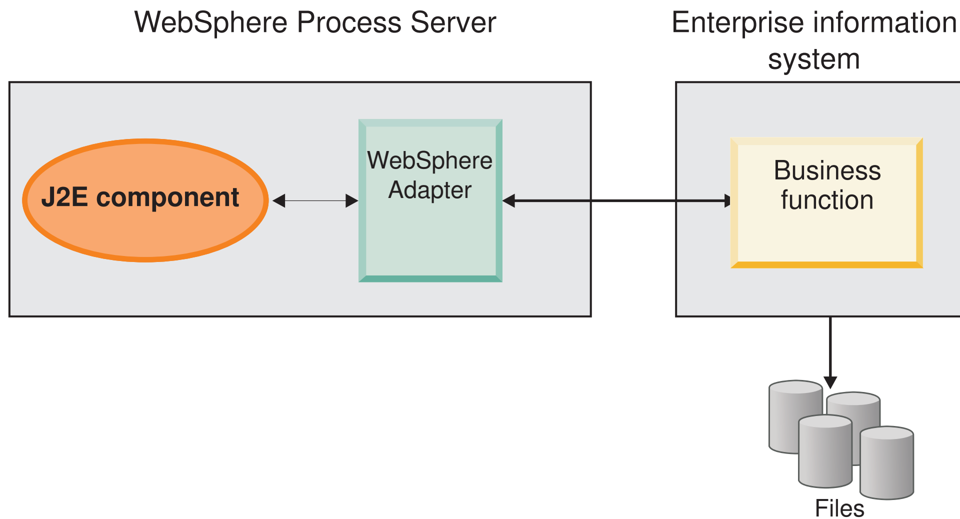


Figure 1. A WebSphere Adapter

The IBM WebSphere Adapter portfolio of adapters is based on the Java 2 Platform, Enterprise Edition (J2EE) standard. JCA is a standard architecture for integrating J2EE applications with enterprise information systems. Each of these systems provides native APIs for identifying a function to call, specifying its input data, and processing its output data. The goal of the JCA standard is to provide an independent API for coding these functions, to facilitate data sharing, and to integrate J2EE applications with existing and other enterprise information systems. The JCA standard accomplishes this by defining a series of contracts that govern interactions between an EIS and J2EE components within an application server.

Fully compliant with the JCA standard, WebSphere Adapters have been developed to run on multiple server runtimes. A WebSphere Adapter does the following:

- Integrates with multiple server runtimes, including WebSphere Process Server, WebSphere Enterprise Service Bus, and WebSphere Application Server.
- Connects an application running on WebSphere Process Server with an EIS.
- Enables data exchange between the application and the EIS.

Each WebSphere Adapter is made up of the following:

- An implementation of the (J2EE) Connector Architecture (JCA), version 1.5, which supports WebSphere Process Server and WebSphere Application Server.
- An enterprise metadata discovery component.

  You use this component with the external service discovery wizard to introspect the EIS to discover and generate business objects and other service component architecture (SCA) artifacts that are compiled in a standard enterprise application archive (EAR) file. You can also use the external service discovery wizard to build a service (rather than discover a service). For example, you can use the enterprise service discovery wizard to build services (based on search criteria you provide), and generate business objects and interfaces.

  The enterprise metadata discovery component implements version 1.1 of the enterprise metadata discovery specification.

WebSphere adapters utilize a format-independent data model for representing data objects. In a WebSphere Process Server or WebSphere Enterprise Service Bus runtime environment, WebSphere adapters use an extension of the service data objects (SDO) for representing data objects.

Architectural overview

In conjunction with the appropriate EIS-specific subclasses, the WebSphere Adapter Foundation Classes provide a JCA-compliant resource adapter implementation that can be managed by the application server to enable bidirectional connectivity to an enterprise information system (EIS).

Outbound requests, those requests intended for the EIS, can be sent to the resource adapter by any J2EE component via the Common Client Interface (CCI) defined by the JCA specification. For inbound events, events within the EIS sent to the adapter, message-driven beans that implement the `InboundListener` interface are registered with the adapter by the application-server enabling them to receive any appropriate inbound events from the EIS via the adapter.

Regardless of the whether data is intended for inbound or outbound delivery, the resource adapter (i.e. Adapter Foundation Classes plus EIS-specific subclasses) acts as a conduit for any J2EE application to communicate with the EIS.

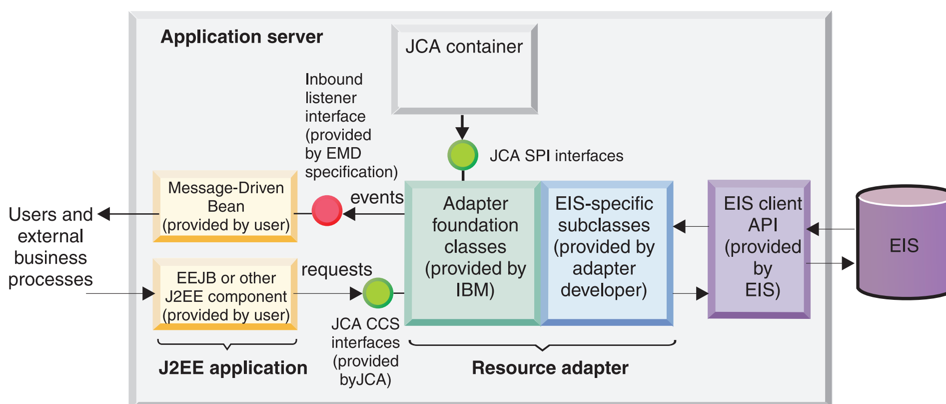


Figure 2. Architecture overview

Runtime architecture component model

The adapter runtime architecture is a collection of components interacting through well-defined interfaces and based on the J2EE Connector Architecture (JCA) specification version 1.5.

At run time, the adapters implement both the Data Exchange Service Provider Interface (DESPI) contracts to interact with the runtime container and the CCI contracts to interact with the application component. The JCA architecture has been extended with the data exchange component that allows efficient data exchange between the adapter and the server runtime and supports multiple runtime environments.

The layered approach provides a set of elements that can be assembled to provide desired functionality and quality of service. The componentized approach allows the separate, independent development of each element, as well as their reuse. The following illustration presents the runtime architecture component model. The

processing performed by each component (and subcomponent) in the model is described in sections that follow the illustration.

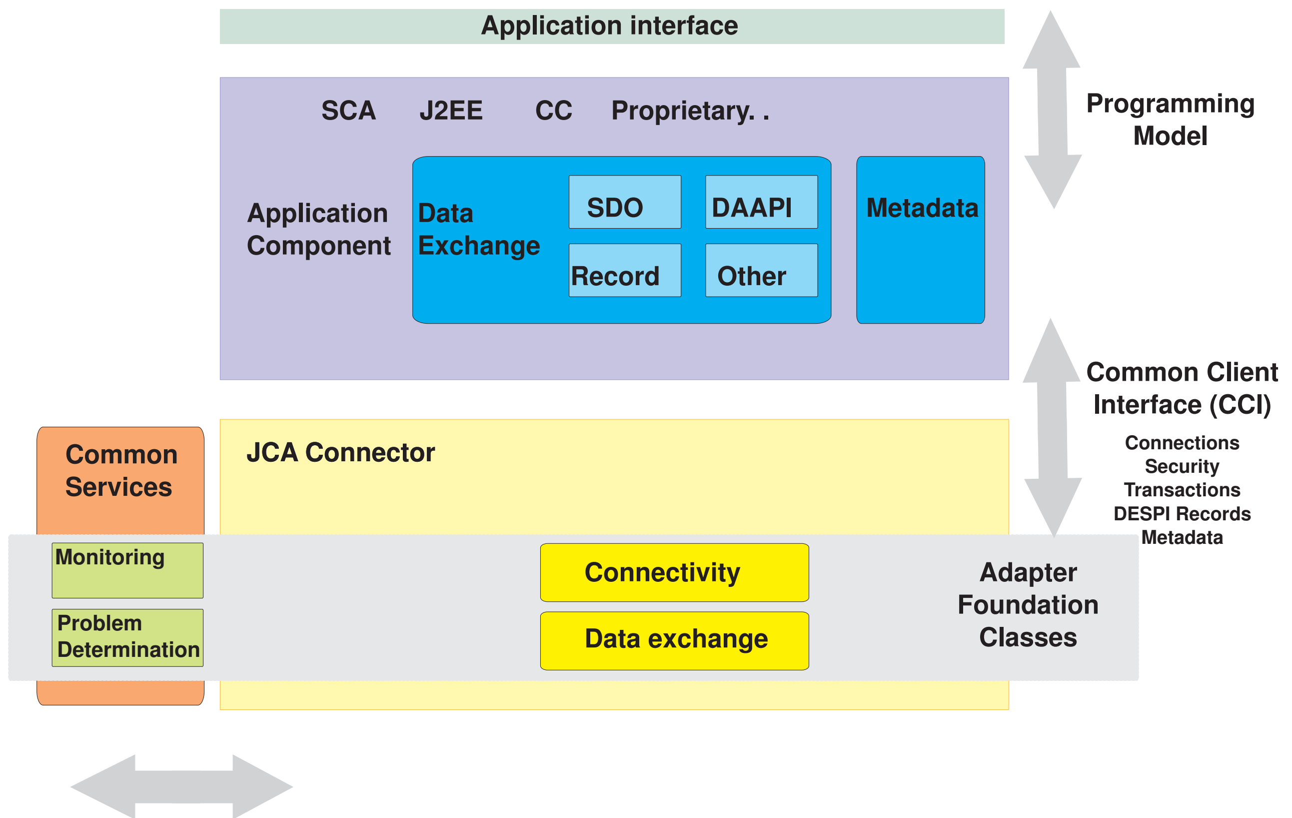


Figure 3. Runtime architecture component model

The component model allows for a single architecture for adapter development and evolution. It uses recognized standards but extends them as necessary, for example with high performance, runtime-independent data exchange interface that enables full adapter functionality in various environments. The layered component model reduces complexity and allows extensibility. The use of the adapter foundation classes (the standard library for building adapters) supports common adapter capabilities and ensures adapter consistency.

JCA connector component

The JCA connector component provides the standard architecture for connecting the J2EE platform to heterogeneous enterprise information systems (EIS), facilitating bidirectional data exchange with the EIS. The JCA connector component can be driven by the metadata that describes the interaction with the EIS. The JCA connector component includes separate subcomponents for connecting to the EIS and for exchanging data.

JCA connector connectivity subcomponent

The connectivity subcomponent of the JCA connector component includes functionality for establishing, maintaining and closing connections to the target EIS and provides support for transactions and security for these connections.

The connectivity subcomponent interacts with the target enterprise information system's specific libraries and functionality. The subcomponent is exposed to the application interface component through standard JCA CCI interfaces, which include Connection, ConnectionFactory, Interaction and InteractionSpec for outbound processing and ResourceAdapter and ActivationSpecWithXid for the inbound event processing.

JCA connector Data exchange SPI (DESPI) subcomponent

The data exchange subcomponent of the JCA connector component includes functionality for sending or receiving data from the application component through the data exchange interface (DESPI). This interface is format and runtime neutral, and permits any kind of structured or streamed data to be moved between the connector and the application. The connector component understands the data format of the EIS and is able to convert it to invocations of Data Exchange SPI. The main advantage of DESPI is its high efficiency rate of passing data between components without introducing any intermediate format.

Common services and adapter foundation classes

The adapter foundation classes provide base adapter implementation ensuring that all the interfaces required by the contracts supported by the Connector component are provided.

As shown in Figure 3 on page 5, the adapter foundation classes provide support across all the elements of the connector component, including the JCA interfaces, as well as data exchange interfaces implementing Data Exchange SPI (DESPI) and various quality of service.

The adapter foundation classes support all the required JCA contracts for outbound and inbound connectivity. For the outbound interactions, the support includes standard create, retrieve, update, and delete (CRUD) operations through the Command Manager, the application sign-on and transactions when supported by EIS. For the inbound connectivity, support includes reliable event delivery to the endpoint, support for polling as well as event listening pattern.

The adapter foundation classes provide full support for DESPI with full implementation of required interfaces including abstract representation of the metadata. An important component of the runtime environments, and thus architecture, is monitoring and problem determination support. The adapter foundation classes provide a set of utility classes for robust and consistent logging and tracing in different deployment scenarios by hiding the underlying runtime implementation.

The adapter foundation classes also support various monitoring and events and allow all adapters take advantage of that functionality. The supported quality of service includes Performance Monitoring Infrastructure (PMI), Application Response Measurement (ARM) and Common Event Infrastructure (CEI). For more information on the monitoring capabilities available, see "Monitoring and measuring performance" on page 181.

Application interface component

The application interface component bridges the runtime environment and the connector component. It enables invocation of the connector from the clients, using

the appropriate programming model. It is responsible for mapping the specific invocation to the invocation of the connector component through the JCA common client interface (CCI).

The component developer who has knowledge of the connector invocation interfaces and the runtime programming model, delivers the application component. The application component consists of data exchange, application interface, and metadata elements.

Metadata

The metadata subcomponent describes the structure of the data exchanged with the EIS through the data exchange interfaces. The metadata can also provide information to the connector component about how the data can be exchanged with the EIS. For more information on the metadata subcomponent, see "How metadata is used at build time and run time."

How metadata is used at build time and run time

Metadata is a set of characteristics that describe the structure of a WebSphere business integration component, such as a business object, collaboration, or adapter. Metadata describes facets common across an entire class of objects. For example, attributes, properties, verbs, and *application-specific information* constitute the metadata for a business object. Application-specific information is the part of metadata of a business object that enables the adapter to interact with its application or a data source.

At build time, you use the adapter to access data on an EIS, and from that data to generate metadata in the form of annotated XML schemas. This build time representation of the metadata contains the annotations with application-specific information (ASI) that the adapter needs at run time in order to associate objects or individual properties with their equivalent in the external resource.

The application-specific information portion of metadata identifies key fields, mappings to external types, or for any other purpose that the adapter dictates. At run time, an appropriate runtime representation of the metadata is passed to the adapter. The adapter generates the data as XML annotations. All other components apart from the adapter itself ignore the XML annotations. The runtime representation must correspond to the XML schemas that were the result of the import during enterprise metadata discovery, however their method of creations may be specific to the given environment.

Note: The XML Schema generated as part of Enterprise Metadata Discovery can be thought of as the canonical form.

You may prefer to construct the data object metadata programmatically from its own metadata representation, which would have been created from the original XML schemas, in other cases different representations, such as Records are used. However, it is required that any application-specific information annotations in the schemas be preserved by such environments and then provided along with the type definition for consumption by the adapter.

Using Enterprise Metadata Discovery to build services

In addition to using enterprise metadata discovery to discover existing services, you can use it to build services that include integrated processes that exchange information with a technology like email or a file system.

Version 1.1 of Enterprise Metadata Discovery includes enhancements for configurable data handlers, function selectors, and data bindings, and a way to *build* service descriptions using these configured artifacts and existing schemas. For information on implementing interfaces for technology-style adapters, see "Enterprise Metadata Discovery interfaces and implementation for technology adapters" on page 157.

IBM WebSphere Adapter Toolkit overview

WebSphere Adapter Toolkit contains everything you need to create a resource adapter. The adapters are metadata-driven components designed for bidirectional communication with external services on Enterprise Information Systems (EIS), such as transaction systems or ERP systems, as well as bidirectional communication with technologies such as Email or Flat File.

Developing application components or processes that interact with the EIS through an adapter requires tools that allow you to discover the services available on the EIS. When the EIS does not include a metadata repository, the tool should allow you to build the appropriate interactions with the EIS using adapters. The enterprise metadata discovery (EMD) specification implemented by the WebSphere Adapters enables you to discover services from an existing metadata repository or build the appropriate interactions with an EIS. The specification defines the interaction between adapters and tools and allows for plugging adapters into a compatible tool implementation that supports the specification.

The enterprise metadata discovery-compliant tools are based on the Eclipse platform and include IBM WebSphere Integration Developer and IBM Rational Application Developer. The current tools support generation of SCA and J2EE programming model artifacts, and is extensible to also support generation of artifacts for other programming models and server runtimes. Enterprise metadata discovery provides *pluggability* for artifact writers, which store the metadata and configuration information in a manner compatible with the tools and runtime requirements. The current enterprise metadata discovery implementation supports SCA artifacts, but other artifacts can be generated by implementing a writer plug-in. Such implementations may choose to write the configuration information and metadata directly to their native repository, or to create classes or configuration files to contain properties required for runtime.

In the service discovery mode, you use the adapter to connect to the EIS metadata repository and browse its contents. You can select objects or services required by the business application from the repository, specify properties for the objects and import them (as business objects) into the development environment.

The enterprise metadata discovery process defines an abstract representation of the imported services that allows the external service wizard to generate artifacts that are specific to the programming model and the supported server runtime.

If you need to build a service (rather than discover a service), you can use the external service wizard to create services from existing artifacts in the context of the particular adapter to be used at run time. You can select the existing data types to be exchanged and defined the transformation to be performed on these data types.

The toolkit includes the following:

- **An adapter development wizard and editor** - Both are Eclipse plug-ins targeted for use with WebSphere Integration Developer, v. 6.2:

– **The New Connector Project Wizard** - Prompts you to specify information about the resource adapter you wish to develop, and then generates code and a deployment descriptor.

  The code generated by the wizard can include sequence of calls, log and trace messages and comments.

– **Resource Adapter Deployment Descriptor Editor** - An Eclipse multi-page form editor that allows you to display and configure your deployment descriptor. As changes are made to configuration properties using the editor, the appropriate Java bean properties are added to your code.

- **Adapter Foundation Classes** - A common set of services for all IBM WebSphere resource adapters.
- **Samples** - To assist you in creating custom WebSphere resource adapters.

New Connector Project wizard overview

The wizard installed with the WebSphere Adapter Toolkit guides you through creation of a Connector Project, including the generation of code and a deployment descriptor for a custom JCA resource adapter.

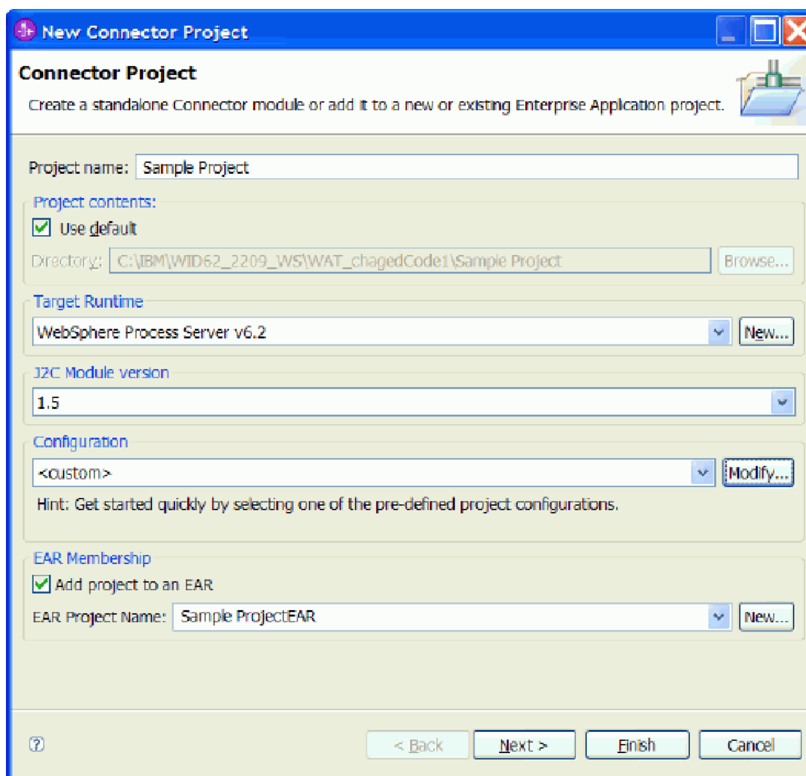


Figure 4. New Connector Project wizard

The New Connector Project Wizard is an Eclipse plug-in intended for use with WebSphere Integration Developer. The wizard allows you to create code in a Java Connector Project for a custom resource adapter. Working with the wizard, you can do the following:

- **Generate implementations of the JCA 1.5 interface specification or extensions to the Adapter Foundation Classes API**

  The wizard prompts you for information that is then used to create the appropriate code for your adapter.

- **Generate a resource adapter deployment descriptor**

  You can view and edit this deployment descriptor using the Resource Adapter Deployment Descriptor Editor.

Resource Adapter Deployment Descriptor Editor overview

This multi-page editor allows you to display, configure, and validate the resource adapter deployment descriptor generated by the wizard.

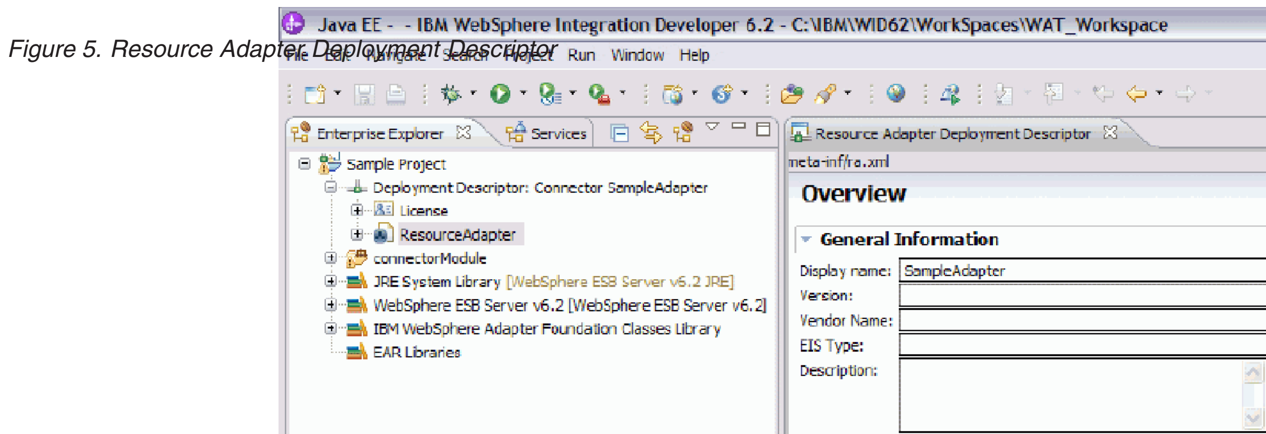


Figure 5. Resource Adapter Deployment Descriptor

The Resource Adapter Deployment Descriptor is an Eclipse plug-in intended for use with WebSphere Integration Developer. The editor allows you to do the following:

- **Display and configure the resource adapter deployment descriptor** without having to modify the XML file directly
- **Automatically generate Java bean properties** in the generated source code that correspond to the configuration properties you add using the editor
- **Validate the deployment descriptor** against the against the JCA 1.5 deployment descriptor schema.

Adapter Foundation Classes overview

The Adapter Foundation Classes installed with theWebSphere Adapter Toolkit provide the foundation services for a custom JCA resource adapter that can run on multiple server runtimes, including WebSphere Process Server.

The New Connector Project wizard uses the Adapter Foundation Classes to generate an implementation of classes for your custom adapter. The Adapter Foundation Classes conform to, and extend, the Java 2 Connector Architecture JCA 1.5 specification. The foundation classes include generic contracts and methods to develop a working resource adapter. The New Connector Project wizard collects information from you to create enterprise information system-specific extensions to the foundation classes.

This document contains implementation guidelines for the Adapter Foundation Classes. The Javadoc for the Adapter Foundation Classes, as well as the Javadoc for the Data exchange service provider interface (DESPI) and the Javadoc for Enterprise Metadata Discovery, are included as part of the WebSphere Adapter Toolkit installation. For more information, see “IBM WebSphere Adapter Toolkit tasks” on page 11.

IBM WebSphere Adapter Toolkit tasks

The tasks range from installing the toolkit, samples, and Adapter Foundation Classes (using the Eclipse Update Manager in WebSphere Integration Developer) to implementing and validating your code.

Table 1. WebSphere Adapter Toolkit tasks

Task	Task description
Validate WebSphere Adapter Toolkit installation requirements	See the developerWorks site for WebSphere Adapter Toolkit for hardware and software prerequisites for the toolkit and for specific information on how to integrate the toolkit into the version of WebSphere Integration Developer or Rational® Application Developer for WebSphere Software that is installed on your system. **Note:** Make sure you select the tab for version 6.2 of the WebSphere Adapter Toolkit.
Installing WebSphere Adapter Toolkit	You do not run an installer program for the toolkit. Instead, you bring the toolkit, the samples, and the adapter foundation classes (including the Javadoc for the Adapter Foundation Classes and the Javadoc for DESPI and EMD) into WebSphere Integration Developer by launching the Eclipse Update Manager.
Access the Javadoc for the Adapter Foundation Classes, EMD 1.1 and DESPI.	To access the Javadoc, perform the following steps: 1. From the Rational Application Developer for WebSphere Software or WebSphere Integration Developer menu, select **Help → Help Contents** 2. Choose **Websphere Adapter Toolkit documentation** and then select the Javadoc you want to view. **Note:** To view the Javadoc for JCA 1.5, go to the download section of the J2EE Connector Architecture site.
Using the New J2C Resource Adapter Project wizard	How to launch it, specify properties, choose generation options, and generate classes.
Using the Resource Adapter Deployment Descriptor editor	How to launch it, display features, change and add properties, and edit source.
Implementing code stubs	A topic-by-topic guide to implementing the generated code.
Validating code	How to validate your code.
Exporting the resource adapter	How to export a standalone (RAR file) or embedded (EAR file) resource adapter.

IBM WebSphere Adapter Toolkit installation requirements

The WebSphere Adapter Toolkit installation has operating system requirements and hardware requirements.

Among the requirements are a Windows or Linux operating system and successful installation of WebSphere Integration Developer.

Operating system requirements

Make sure that you meet the operating system requirements shown in the table.

Operating system	Versions
Linux	Red Hat Enterprise Linux AS/ES/WS 3 Update 4, Version 3.0 SUSE LINUX Enterprise Server (SLES and SLSS), Version 9.0

Operating system	Versions
Windows 2000	Windows 2000 Professional (SP4) Windows 2000 Server (SP4) Windows 2000 Advanced Server (SP 4)
Windows XP	Windows XP SP 2
Windows 2003	Windows Server 2003 Standard Windows Server 2003 Enterprise

Hardware requirements

The table shows the hardware requirements for supported operating systems.

Operating system	Hardware requirements
Linux® Windows® 2000 Windows 2003 Windows XP	• Intel® Pentium® III 800 MHZ processor or faster • 1024 x 768 display or higher resolution monitor • Memory: requirements – 768 MB minimum – 1 GB recommended • Disk space requirements: – 3.5 GB minimum to installing software prerequisites – Additional disk space for your development resources. You can reduce the minimum disk space if optional features and runtimes are not installed.

Software requirements

The components that comprise the WebSphere Adapter toolkit are Eclipse plug-ins. You must install WebSphere Integration Developer software before attempting to install IBM WebSphere Adapter Toolkit software plug-ins.

Samples overview

When you installed the IBM WebSphere Adapter Toolkit, two sample resource adapters were placed on your system. The samples installed with WebSphere Adapter Toolkit are a reference for the creation of custom JCA resource adapters.

There are two samples as follows:

- The TwineBall sample is an implementation of a custom WebSphere Adapter based on the Adapter Foundation Classes. This sample is located in the `adapter/twineball` directory of the install location you selected.

  The sample adapter connects to a sample enterprise information system (EIS), which is also called TwineBall. This EIS is included in the RAR package, twineball.jar. The TwineBall EIS uses the Derby database to store table data in a file on the file system *TWINE*.
- The KiteString sample is similar to TwineBall, but is based directly on the JCA 1.5 resource adapter interface specification. The KiteString sample is located in the `adapter/kitestring` directory of the install location you selected.

This documentation describes how to run the Twine Ball sample only. However, you can apply the instructions for running and testing the Twine Ball sample to the Kite String sample.

The sample instructions accommodate creating an adapter for the Twine Ball sample using WebSphere Integration Developer, for deployment to a WebSphere Process Server runtime environment, as well as instructions for creating the adapter for the Twine Ball sample using Rational Application Developer, for deployment to WebSphere Application Server runtime environment.

In order to test the functionality of the adapter used in the sample, the adapter must include both outbound and inbound processing capabilities. In the instructions for the sample, you will first select an outbound processing direction for the adapter. The outbound processing performed by the adapter in the Twine Ball sample creates a customer. As a result of creating a customer, an event is created in a database table. When you test the adapter in the sample, the adapter uses its inbound processing capabilities to retrieve that event.

Both Twineball and KiteString include the following:
- Implementation of a resource adapter and enterprise metadata discovery
- Resource adapter deployment descriptor and source code in Project Interchange Format suitable for importing into WebSphere Integration Developer
- RAR file suitable for deployment to WebSphere Process Server.

Running the Twine Ball sample using WebSphere Integration Developer

Use WebSphere Integration Developer to access and run the Twine Ball sample.

Import the samples code

Before you can run the sample, you must first import it into your workspace.

Make sure you have installed WebSphere Adapter Toolkit.

For information on known issues with regard to running the sample code, see *Known issues* in Troubleshooting the samples.

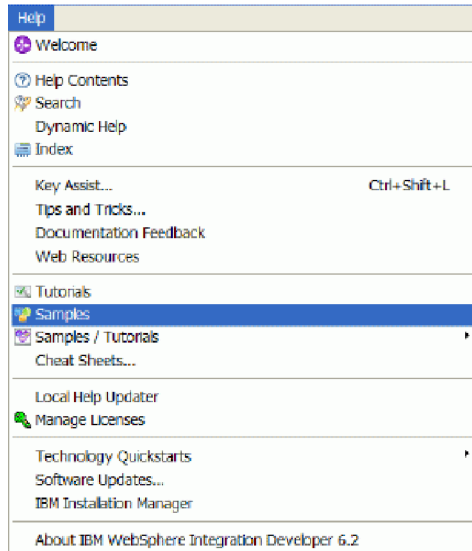
Importing the sample code involves bringing sample code and artifacts into your environment so that you can run a sample application.

The following instructions describe how to use WebSphere Integration Developer to import a deployable RAR file for use in the Twine Ball sample.

Optionally, you can import the Twine Ball sample by importing the source code.

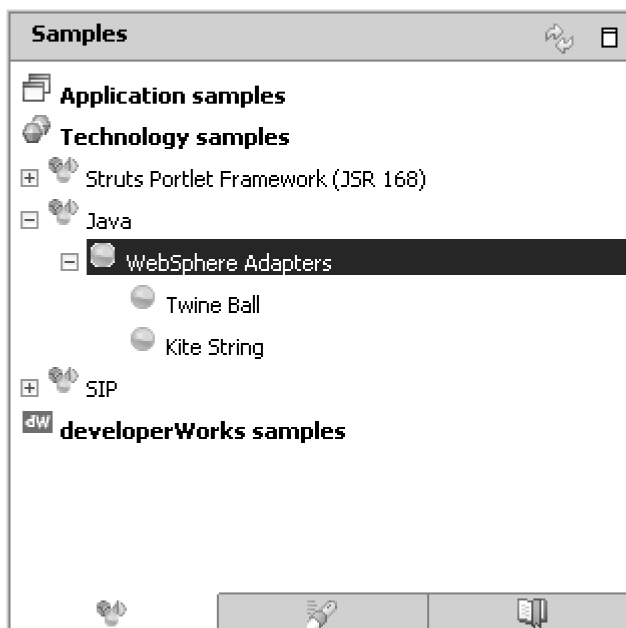
1. Import the deployable RAR file for the Twine Ball sample from the WebSphere Integration Developer samples.
   a. Launch WebSphere Integration Developer

      Click **Start** → **Programs** → **IBM WebSphere** → **Integration Developer 6.2**.
   b. From the menu, select **Help** → **Samples**



This launches the Samples.

c. From the Samples navigation pane, select **Technology samples** and expand **Java** and **WebSphere Adapters** so that the Twine Ball and Kite String samples display.



d. Click **Twine Ball** to display a description of the Twine Ball sample in the viewing pane of the Technology Samples.

2. From the viewing pane of the Technology samples window, select **Import the sample deployable rar**. This launches the Connector Import window.

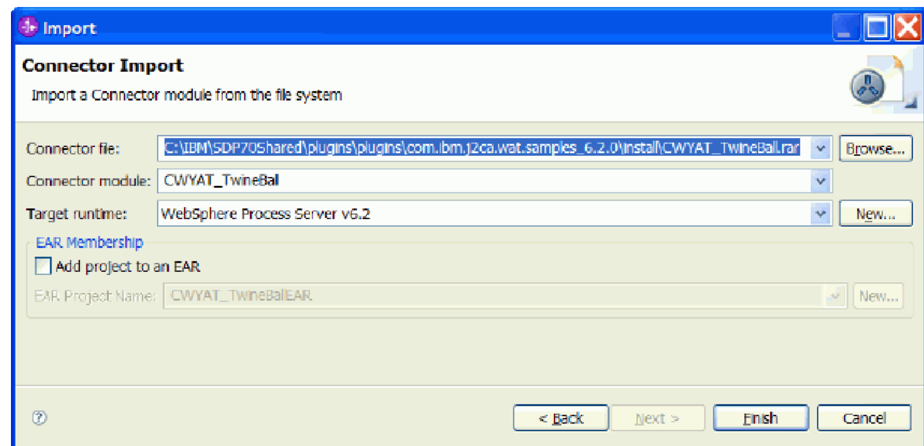
   Note: You can also import the sample deployable RAR from the setup instructions window.

3. Enter values on the Connector Import window:

   a. Accept the default values for the **Connector file** and the **Connector module** fields.

   b. Select `WebSphere Process Server v6.2` for the **Target runtime** field.

c. Optional: Deselect the **Add project to an EAR** check box.



4. Click **Finish**.

   A dialog prompts you to open the J2EE perspective. Click **Yes** to finish the process of importing the deployable RAR file for the Twine Ball sample into your workspace.

Now you can perform external service discovery for the Twine Ball sample.

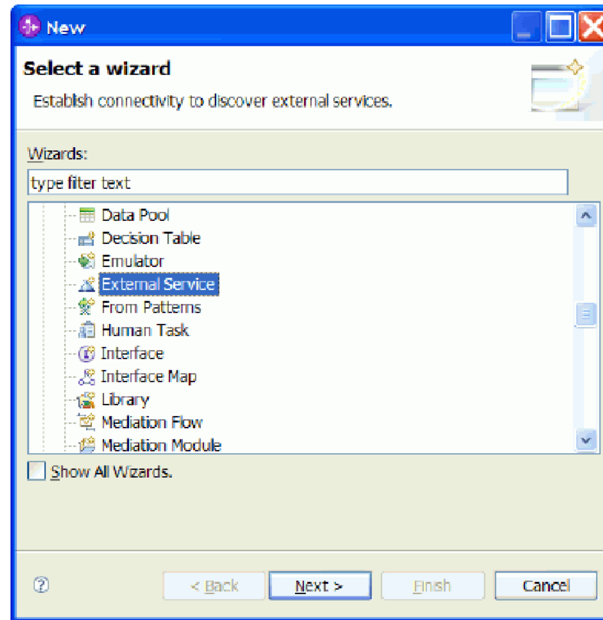
Run external service discovery for outbound processing

The external service wizard is a tool you use to create services. The external service wizard establishes a connection to the EIS, discovers services (based on search criteria you provide), and generates business objects, interfaces, and import or export files, based on the services discovered.

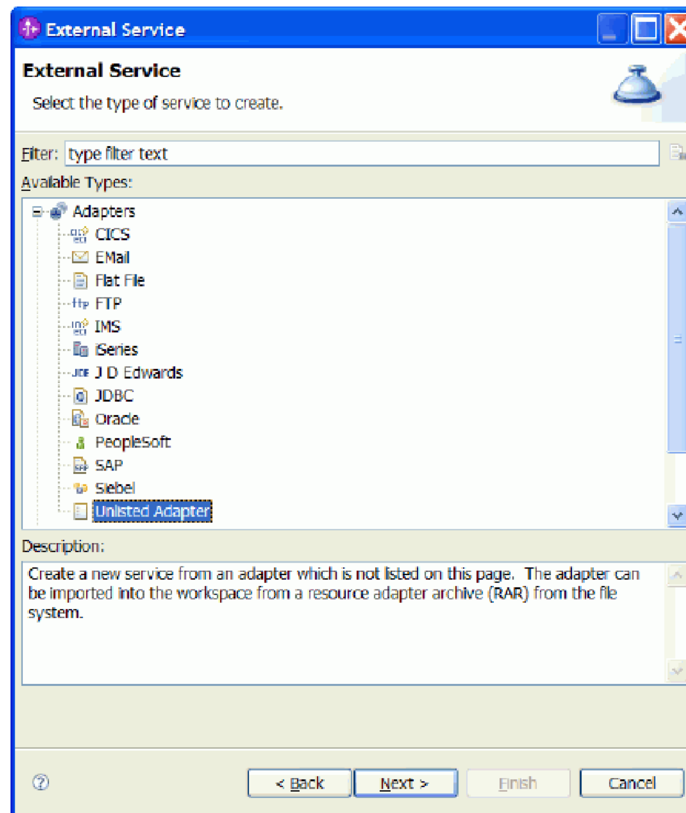
Import the deployable RAR file for the sample.

For the Twine Ball sample you need to create an adapter service for outbound processing and inbound processing. This task describes the how to use the External Service wizard to create and adapter service for outbound processing.

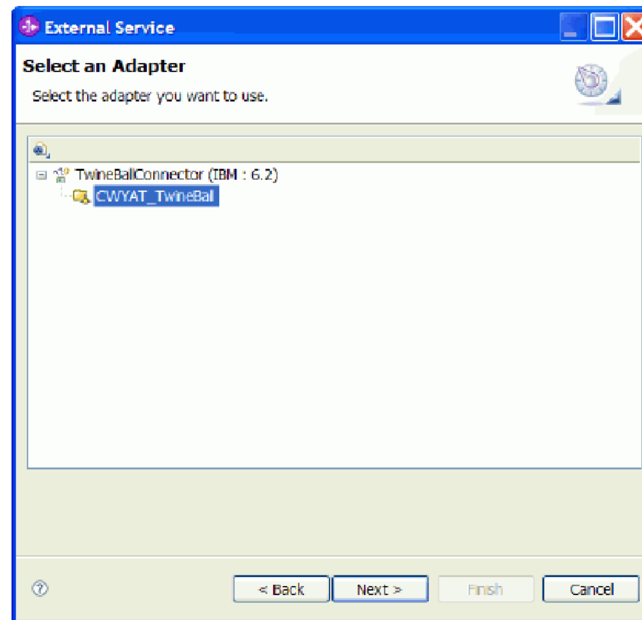
1. If not already there, go to the Business Integration perspective.
2. Place your cursor in the Business Integration navigation pane, right-click and select **New** → **Other** to launch the Select a wizard window.
3. From the list of available wizards, expand **Business Integration** and select the **External Service** wizard:



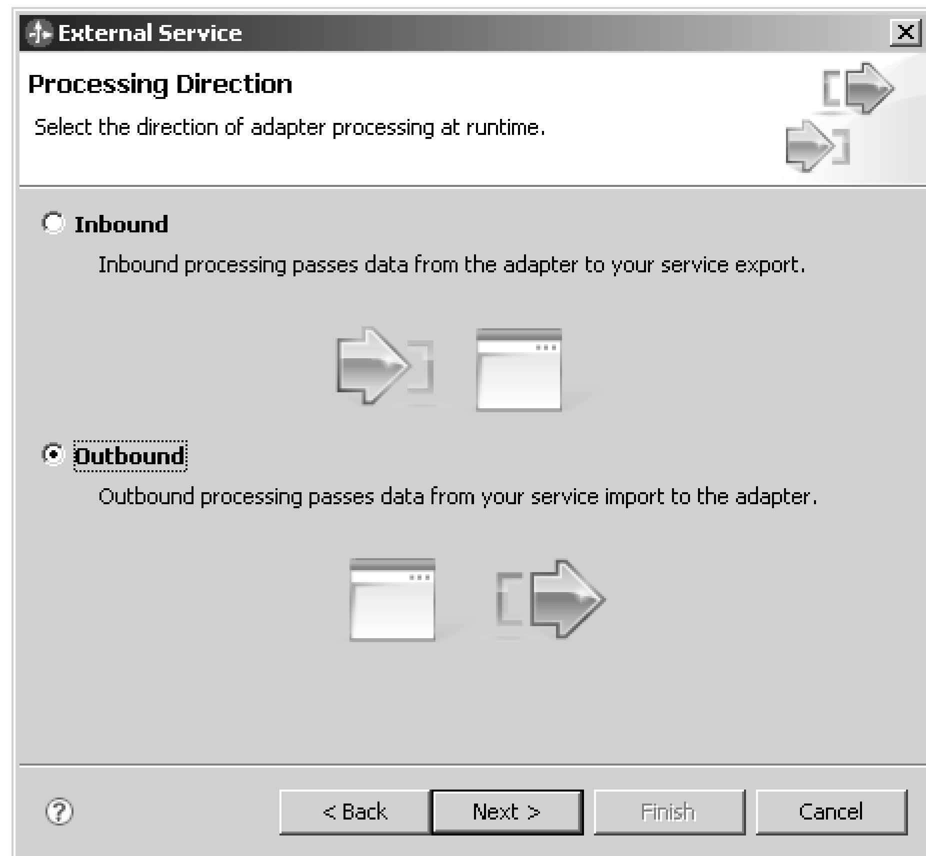
4. Click **Next** to launch the New External Service window.
5. From the New External Service window, make sure that **Unlisted Adapter** is selected and click **Next**.



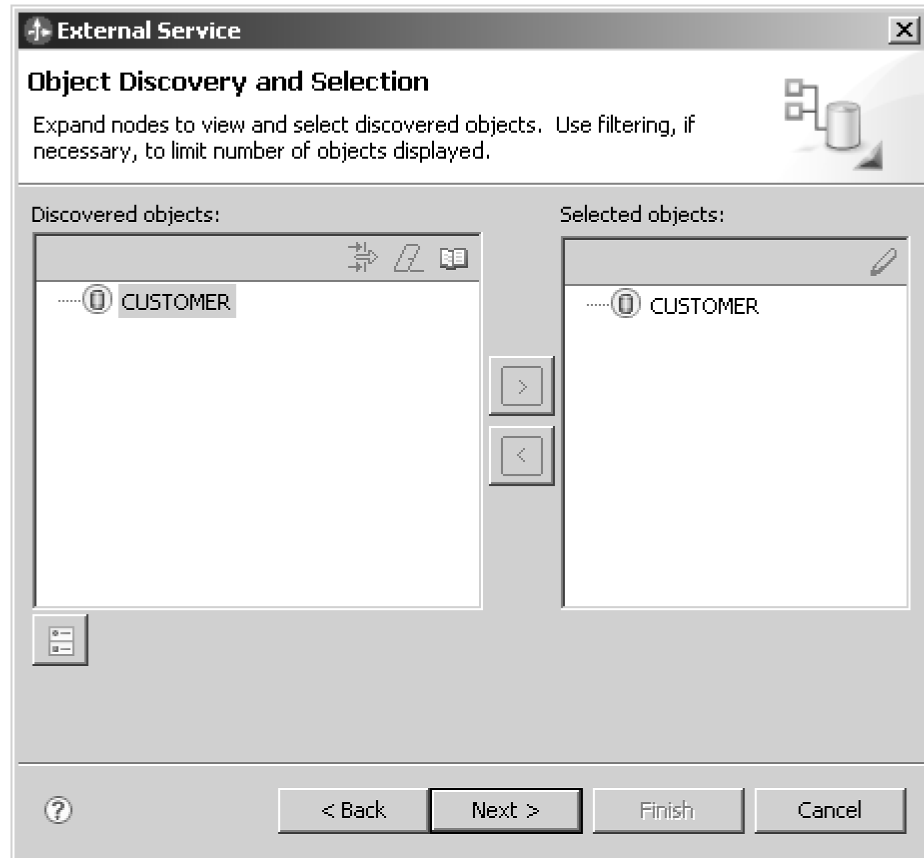
6. From the Select an Adapter window, expand **TwineBallConnector (IBM:6.2)**, select **CWYAT_TwineBall** and click **Next**.



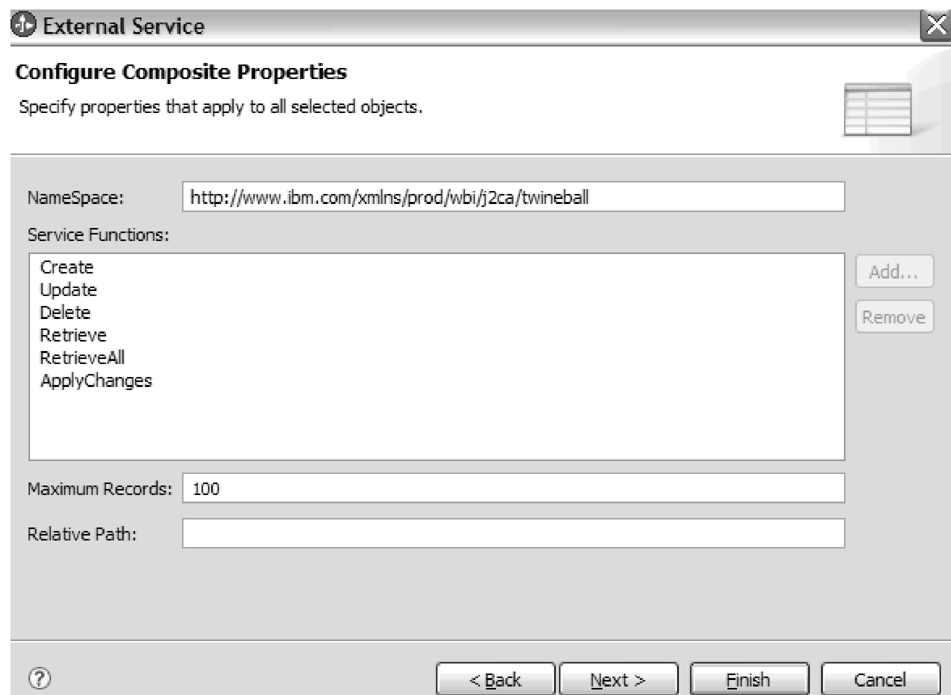
7. From the Processing Direction window, select **Outbound** and click **Next**.



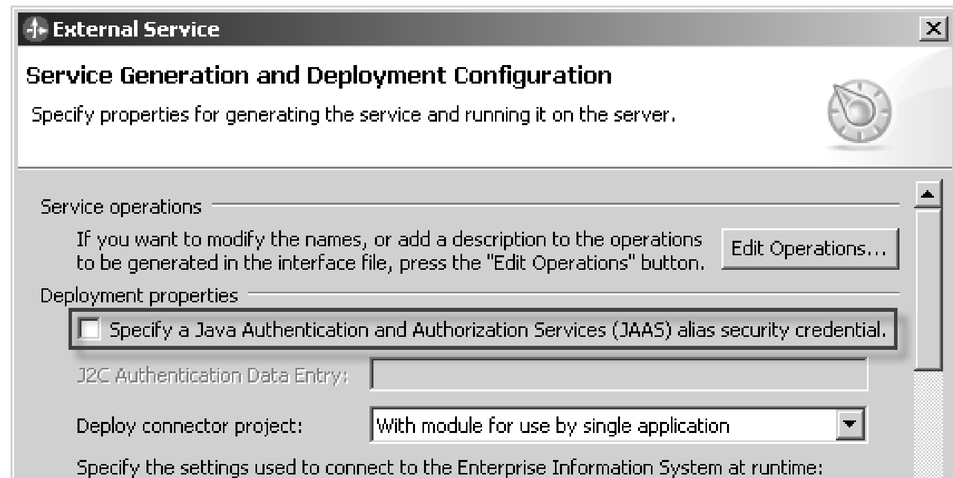
8. From the Discovery Configuration window, click **Next**. No connection properties are required.
9. From the Object Discovery and Selection window, select CUSTOMER from the Discovered objects pane and add it to the Selected objects portion of the window then click **Next**.



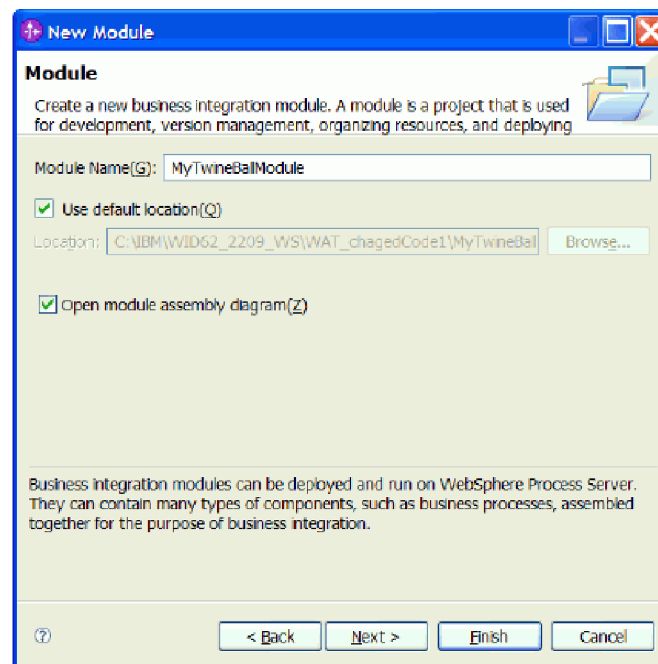
10. From the Configure Composite Properties window, select **Next**.



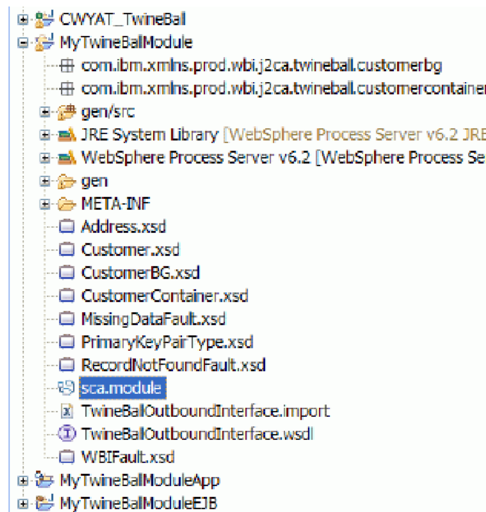
11. From the Service Generation and Deployment Configuration window, deselect **Specify a Java Authentication and Authorization Services (JASS) alias security credential** and click **Next**.



12. From the Service Location Properties window, click **New**. This launches the New Integration Project window.
13. From the New Integration Project window, select **Create a module project** and click **Next**.
14. Enter values in the New Module window and click **Finish**.



Your module displays in the Project explorer view:



15. Click **Finish** from the Service Location Properties window to add the outbound interface to the module.
16. You are prompted on whether you want to update the model, select **Yes**.

Run the external service discovery process again to add the inbound interface to the module.

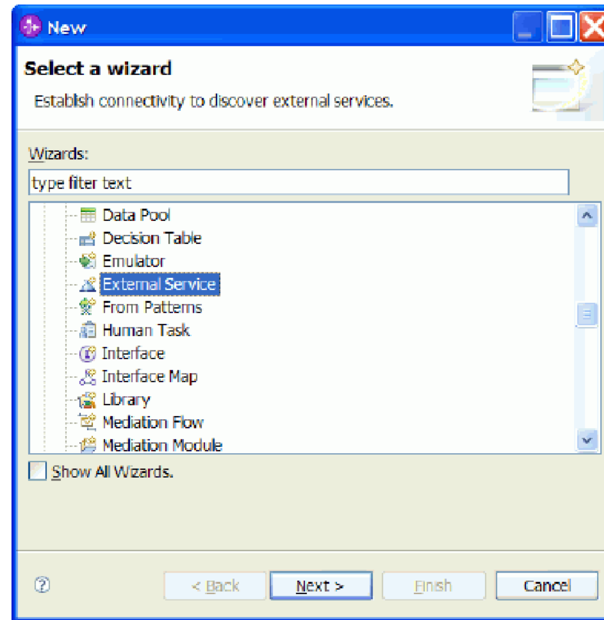
Run external service discovery for inbound processing

The external service wizard is a tool you use to create services. The external service wizard establishes a connection to the EIS, discovers services (based on search criteria you provide), and generates business objects, interfaces, and import or export files, based on the services discovered.

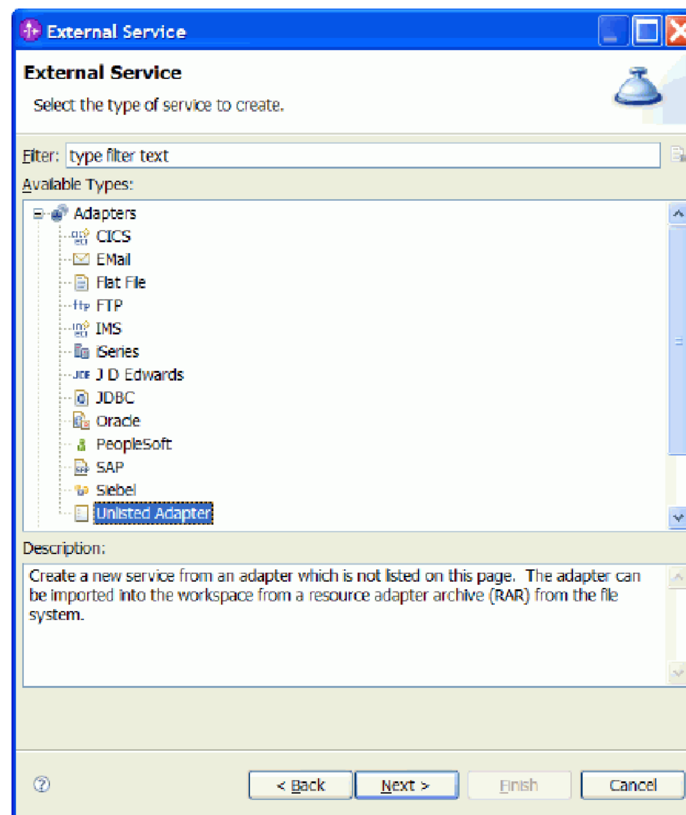
Make sure you have done the following:
- Import the deployable RAR file for the sample into your workspace.
- Run external service discovery for outbound processing.

For the Twine Ball sample you need to create an adapter service for outbound processing and inbound processing. This task describes the how to use the External Service wizard to create an adapter service for inbound processing.

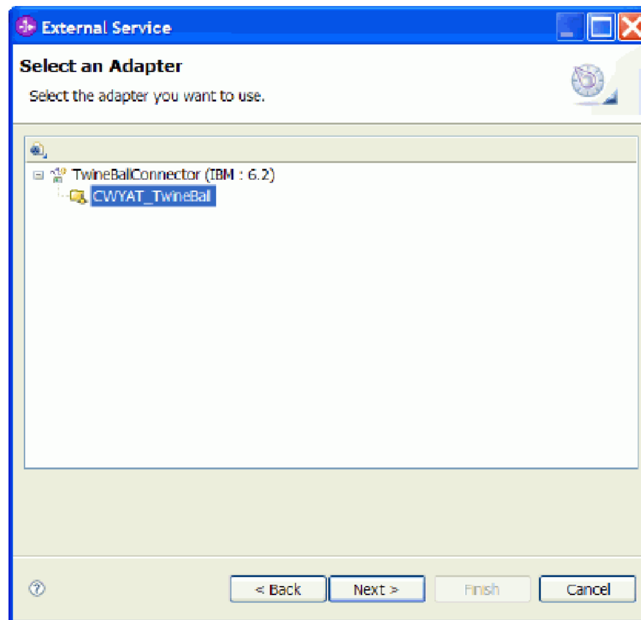
1. Go to the Business Integration perspective.
2. Place your cursor in the Business Integration navigation pane, right-click and select **New** → **Other** to launch the Select a wizard window.
3. From the list of available wizards, expand **Business Integration** and select the **External Service** wizard:



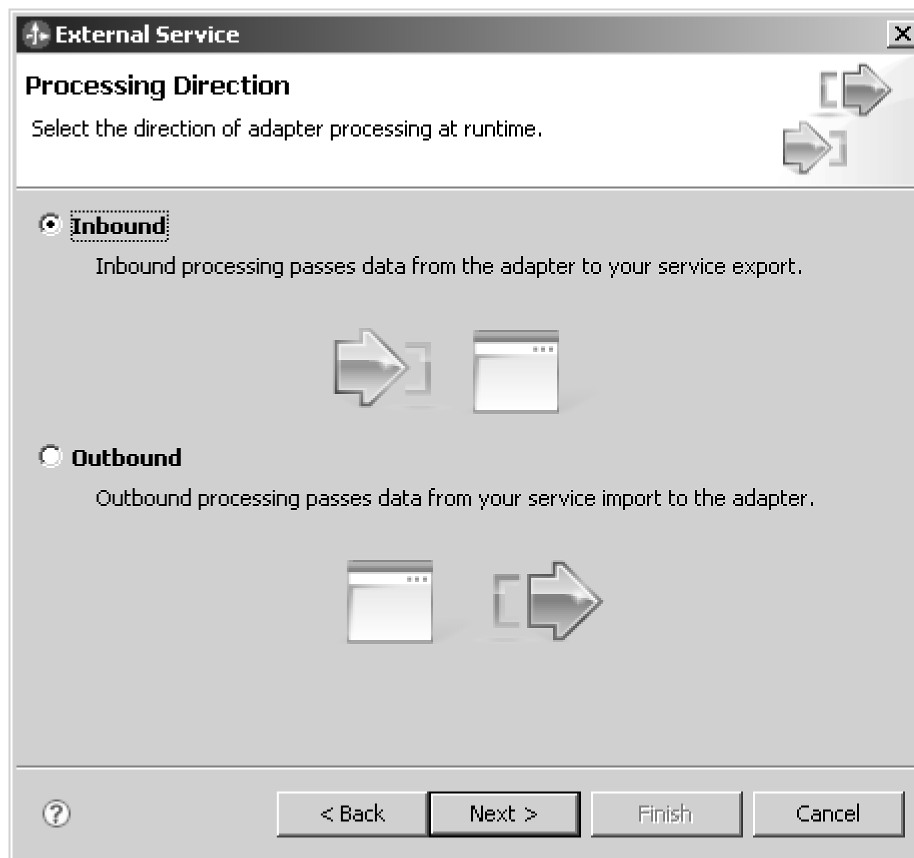
4. Click **Next** to launch the New External Service window.
5. From the New External Service window, make sure that **Unlisted Adapter** is selected and click **Next**.



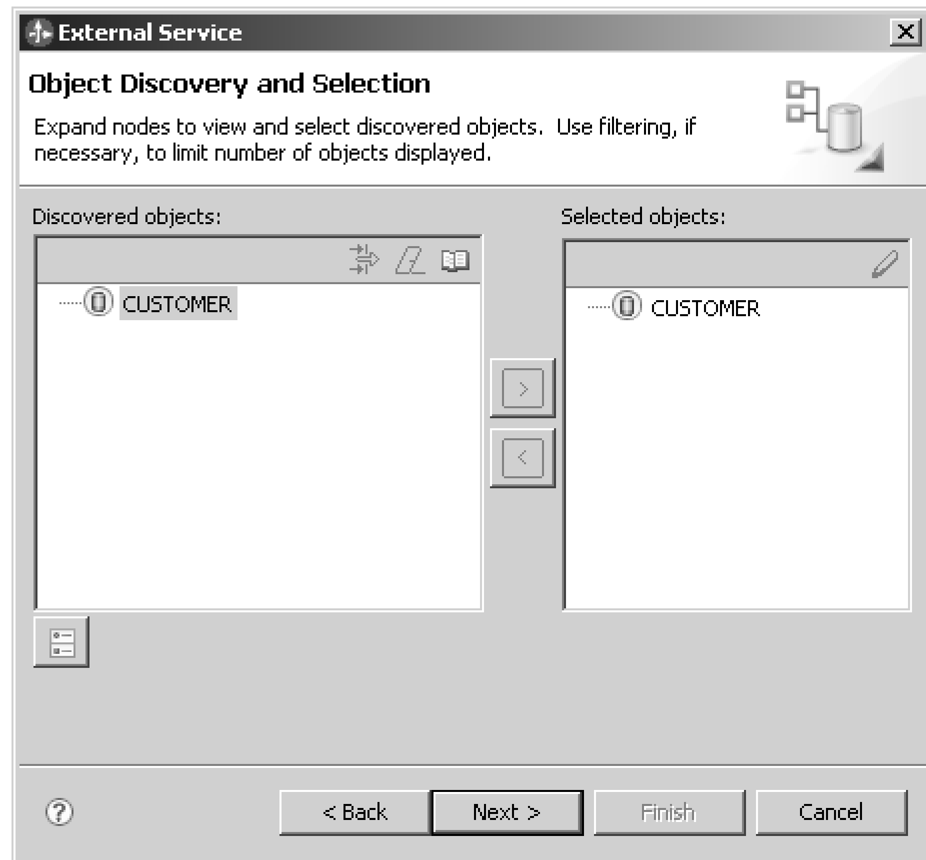
6. From the Select an Adapter window, expand **TwineBallConnector (IBM:6.2)**, select **CWYAT_TwineBall** and click **Next**.



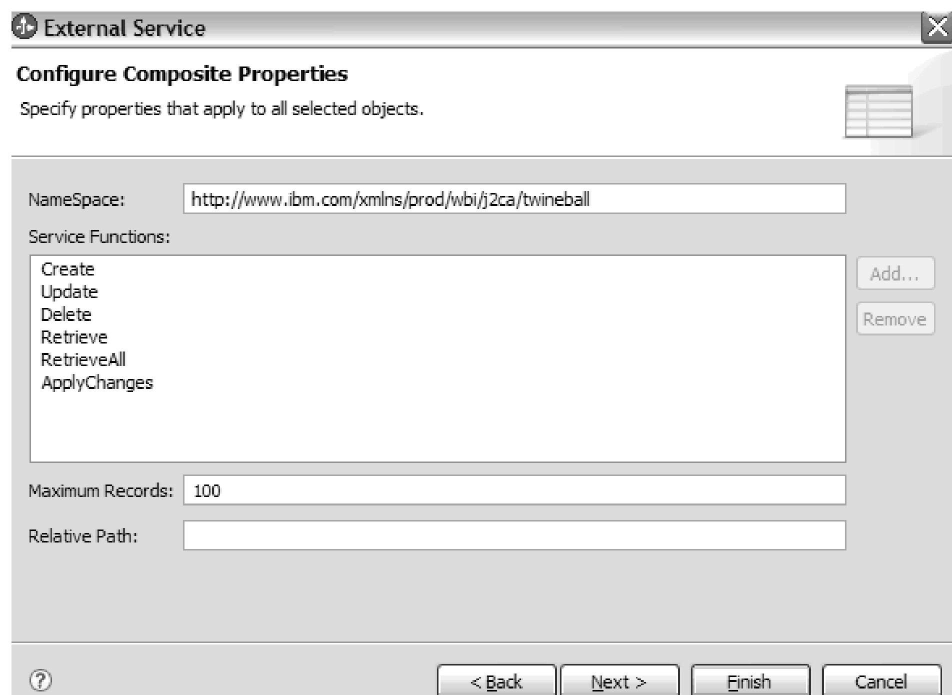
7. From the Processing Direction window, select **Inbound** and click **Next**.



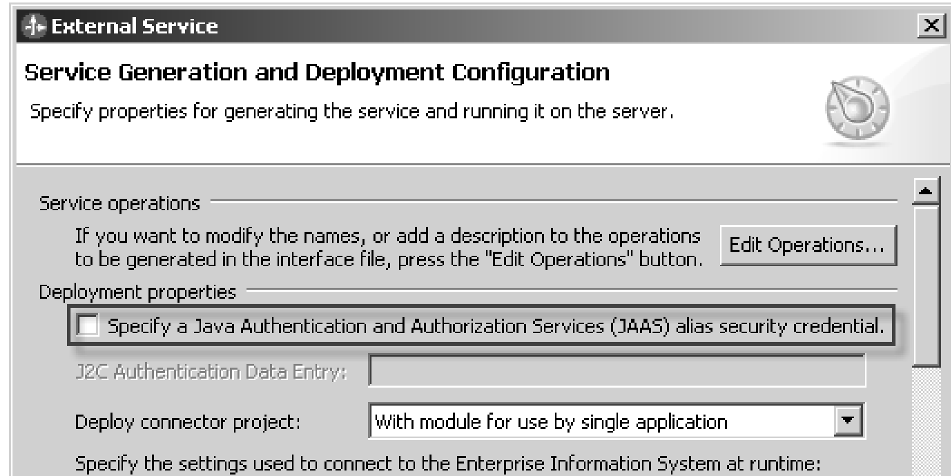
8. From the Discovery Configuration window, click **Next**. No connection properties are required.
9. From the Object Discovery and Selection window, select CUSTOMER and add it to the Selected objects portion of the window then click **Next**.



10. From the Configure Composite Properties window select **Next**.

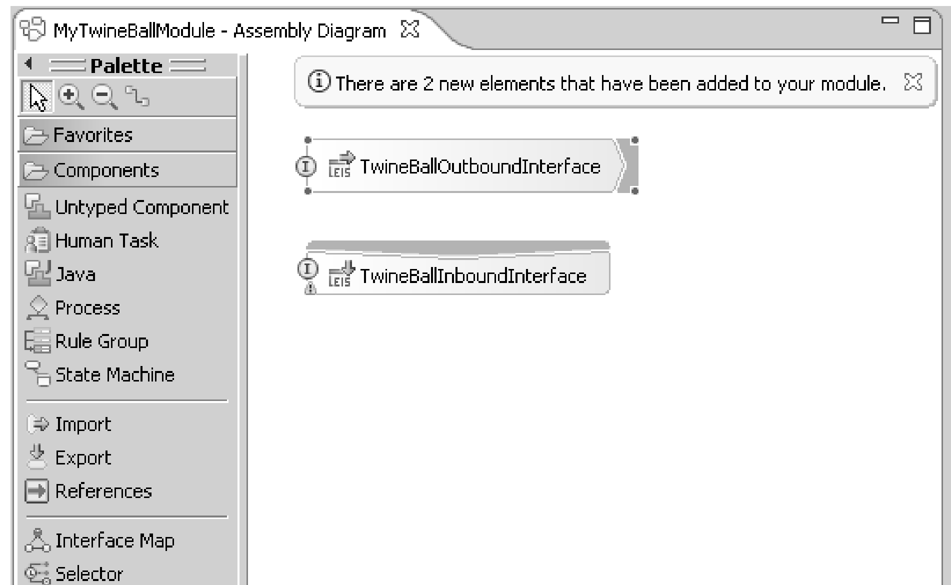


11. From the Service Generation and Deployment Configuration window, deselect **Specify a Java Authentication and Authorization Services (JASS) alias security credential**.



12. From the Service Location Properties window, click **Finish** to add the inbound interface to the module.
13. You are prompted on whether you want to update the model, select **Yes**.

    You should see the inbound and outbound interfaces in the viewing area of the assembly diagram editor:

Modify the module.

Modify the module

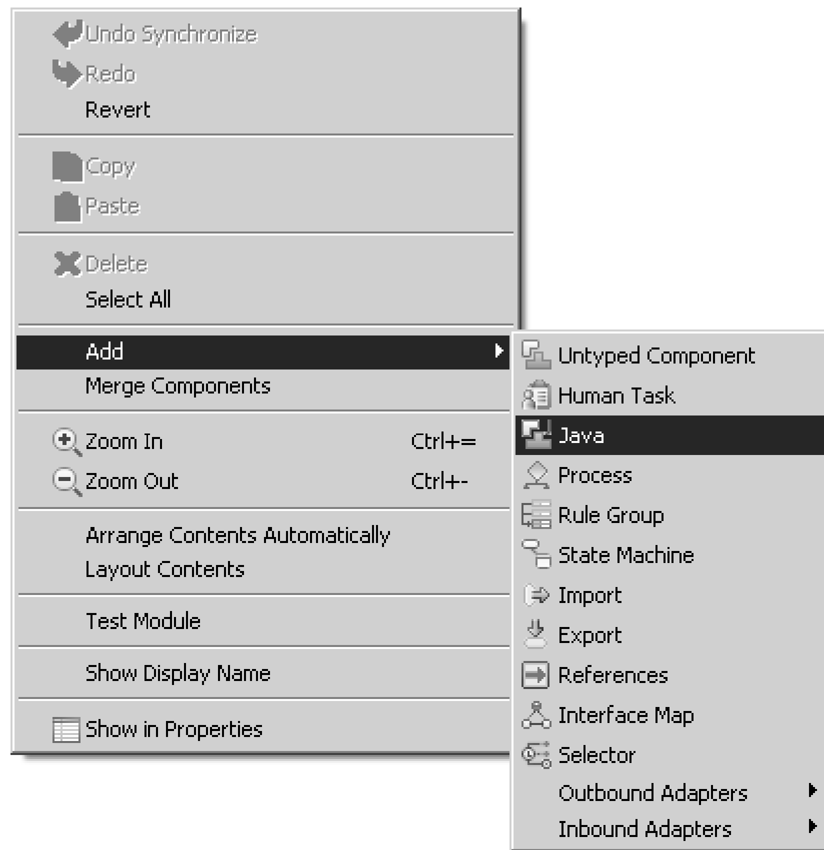
Modify the sample module by adding a Java component to link to the inbound service.

You must have created the service before modifying it.

By adding a Java component to link the service you will be able to access and implement the emit `<Create/Delete/Update>`Customer() method to insert a print statement.

1. Add a Java component to link to the InboundService.

a. In the viewing pane of the Assembly diagram, right click and select **Add** → **Java** to add a Java component to the diagram



2. Add a wire from the inbound interface to the new Java component. A window displays to inform you that your actions will allow the service to be used in other modules. Click **OK**.
3. Right click on the Java component and select **Generate** → **Implementation.** and click **OK** on the Generate Implementation window.
4. Add a `system.out.println` to the emitCreateAfterImageCustomer

```
emitCreateAfterImageCustomer()/emitCustomer() method
   public void emitCreateAfterImageCustomer(
   DataObject emitCreateAfterImageCustomerInput) {
  System.out.println("Got the event!");
  //TODO Needs to be implemented.
 }
```

5. Save the module.

Now you are ready to test the module.

Test the sample

Use the administrative console and test client to test your sample.

Before you test the sample module make sure you have completed all of the previous tasks for creating and modifying the sample module.

1. Start WebSphere Process Server
2. Add the module project to the server
3. Verify the adapter is running (and polling if you selected the inbound service). Check the console, log and trace file for more information.

4. Run the administrative console and verify the module is installed and running.
5. Test the module by performing the following steps:
   a. Change to the Business Integration perspective
   b. Right click on the module and select **Test** → **Test Module**
   c. Populate the customer object fields with data.
   d. Click the continue button.
   e. Look for this message in the console:
      ```
      Got the Event!
      ```
      This is the message that you implemented in the emitCreateAfterImageCustomer() method.

Running the Twine Ball sample using Rational Application Developer

Use Rational Application Developer access and run the Twine Ball sample.

Import the samples code into Rational Application Developer and modify the sample for use

Before you can run the sample, you must first import it into your workspace and modify various values.

Make sure you have installed WebSphere Adapter Toolkit.

For information on known issues with regard to running the sample code, see *Known issues* in Troubleshooting the samples.

Importing the sample code involves bringing sample code and artifacts into your environment so that you can run a sample application.

The following instructions describe how to use Rational Application Developer to import a deployable RAR file for use in the Twine Ball sample.

Optionally, you can import the Twine Ball sample by importing the source code.

1. Import the deployable RAR file for the Twine Ball sample from the Rational Developer samples.
   a. Launch Rational Application Developer
      Click **Start** → **Programs** → **IBM Rational** → **Application Developer 7.5.1**.
   b. From the menu, select **Help** → **Samples**
      This launches the Samples.
   c. From the Samples navigation pane, select **Technology samples** and expand **Java** and **WebSphere Adapters** so that the Twine Ball and Kite String samples display.
   d. Click **Twine Ball** to display a description of the Twine Ball sample in the viewing pane of the Technology Samples.
2. From the viewing pane of the Technology samples window, select **Import the sample deployable RAR**.
3. Right click on **CWYAT_TwineBall** and select **New** → **Other** → **J2C** → **J2C Java Bean**
4. Click **CWYAT_TwineBall** → **Next** → **Next**
5. In the Object Discovery and Selection window use the arrow-right to add the **CUSTOMER** object to the *Objects to be imported* area of the window and click **Next**.

6. In Java Creation and Deployment Configuration, make the following selections:
   - Select **create new project Name** and enter a name for the project, for example, `Demo`.
   - Enter a name in the **Create New Package Name** field, for example `pckg`
   - Provide an interface name in the **Interface Name** field, for example `Sample`
   - Choose **Non-managed Connection** and click **Finish**.
7. Right click on **Demo** → **New** → **Other** → **J2C** and select one of the following:
   - **Web Page**
   - **Web Service**
   - **EJB from J2C Java Bean**

   Select **Next**.
8. Choose **Java Project** and enter a name for the project.
9. In J2C Java bean selection, choose the correct J2C bean implementation and click **Next**. For example, using the values documented in this sample, you would choose:`\Demo\src\pckg\SampleImpl.java`
10. Select **EJB** and click **Next**
11. Create the EJB project name and the EJB package.
12. For example, `DemoEJB` for the project name and `ejbpkg` for the EJB package name.
13. Right click **DemoEJB** → **Properties** and add the following to the Java Build path:
    - twine.jar
    - aspectjrt.jar
    - ffdc.jar
    - ffdcsupport.jar
14. Right click **DemoEJB** → **Java EE** → **Prepare for deployment**
15. Right click on Server and add project If you used values from this example, you would add the project `DEMOEJBEAR`.

Use the universal test client to test the adapter in the Twine Ball sample.

Test the sample using the universal test client

Use the universal test client to verify that the adapter in the Twine Ball sample functions as intended.

Before you test the sample module make sure you have completed all of the previous tasks for creating and modifying the sample module.

The test validates that the adapter performs it's intended functions within the context of the Twine Ball sample.

1. Run universal test client.
2. Click on JNDI Explorer
3. Click on ejb and click on your package, for example `ejbpkg` and select correct **SessionBeanHome**
4. On the left side of Universal test client, choose correct **SessionBeanHome** and Invoke create method. If you used the values from the sample documentation, you would select `EJBSessionBeanHome`.
5. Click the **work with object** radio button.

6. On left side select **EJBSessionBean1** and click **Customer createCustomer(Customer)** Enter input customer parameters

Troubleshooting the samples

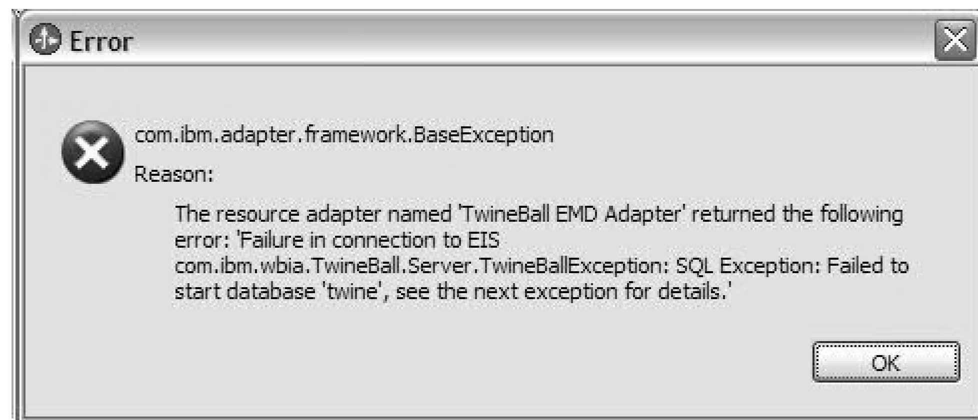
You may need to troubleshoot issues that arise when creating or running the samples.

Errors

The following errors may result when working with the sample code:

- A com.ibm.adapter.framework.BaseException error

  This error can result when using the sample source code to run external service discovery or because the Twine Ball database is locked by a prior run of the external service discovery process. The error message displays as follows:

  To fix this error perform the following steps:

  1. Go to the WebSphere Integration Developer installation directory and delete the **twine** folder.
  2. Restart WebSphere Integration Developer and try to run the external service discovery process again.

  .

Known issues

The following issues regarding working with the Kite String and Twine Ball samples are known issues to development:

- If you create the module using the sample code instead of the deployable RAR file, you cannot publish to the module successfully to WebSphere Process Server. When you run the samples you should do so using the deployable sample RAR file instead of the source code.
- The Kite String sample works in WebSphere Application Server with custom clients, but it does not work with the Rational Application Developer J2C tooling.

Using the New Connector Project wizard

You use the wizard create a Connector Project which contains a deployment descriptor and classes. The classes either extend the WebSphere Adapter Foundation Classes or implement the JCA 1.5 Interface specification.

Launching the New Connector Project wizard

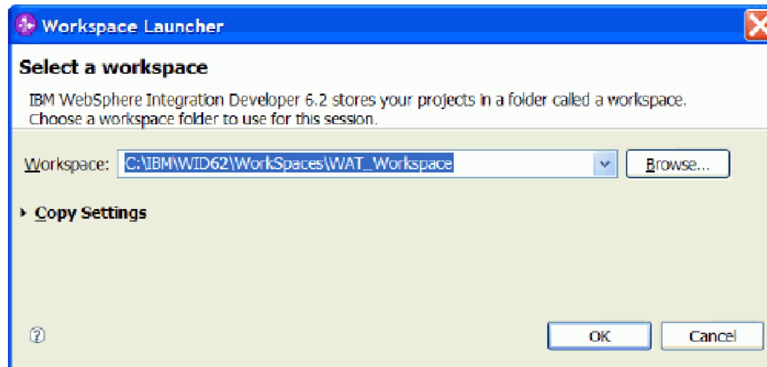
You launch the wizard from IBM WebSphere Integration Developer.

Make sure you have met all of the installation requirements and that you have successfully installed WebSphere Integration Developer and the WebSphere Adapter Toolkit plug-ins.

Launch the New Connector Project wizard when you are ready to create a new adapter project.

1. Start IBM WebSphere Integration Developer.

   Choose **Start** → **IBM WebSphere** → **WebSphere Integration Developer V6.x** → **WebSphere Integration Developer V6.x**. This displays the Workspace Launcher dialog.

   Workspace Launcher dialog

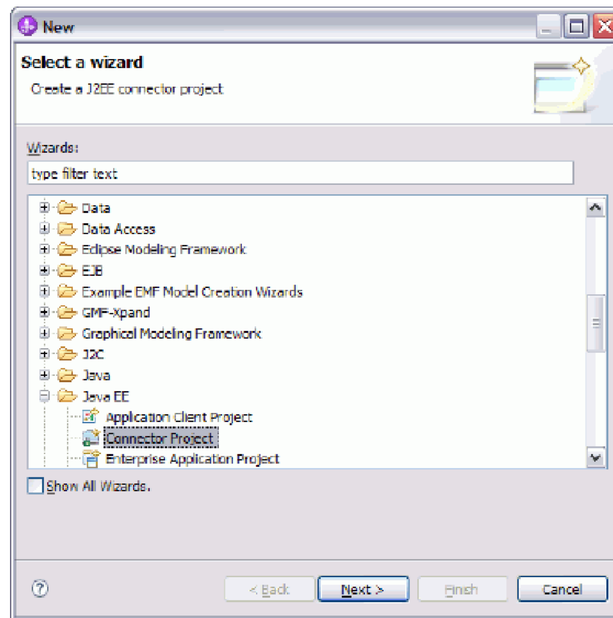
2. Enter a workspace directory for your project, or click **Browse** to select a location for your project in the **Workspace** field.

   The workspace is a directory where WebSphere Integration Developer stores your project.

   Optional: Select the **Use this location as the default** check box to always use this workspace for new projects. You can change workspaces by choosing **File** → **Switch Workspace**.

3. Make sure you are in either the Business Integration or Java EE perspective.
4. Select the wizard.

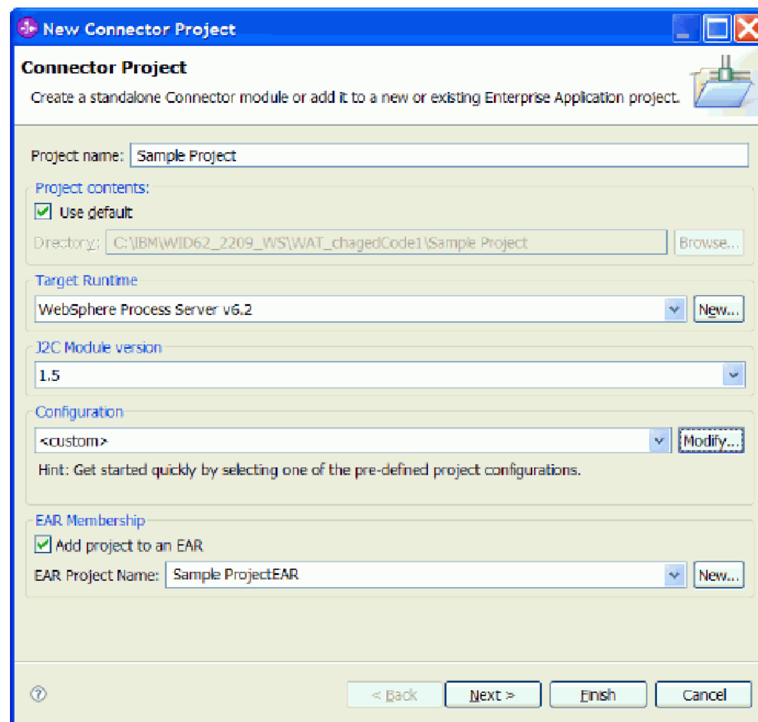
   Choose **File** → **New** → **Other**. This displays the Select a wizard dialog.



Select a wizard dialog

5. Start the wizard.

   Expand the**Java EE** folder, choose **Connector Project**, and click **Next**. This starts the New Connector Project wizard and displays the Connector Project dialog.



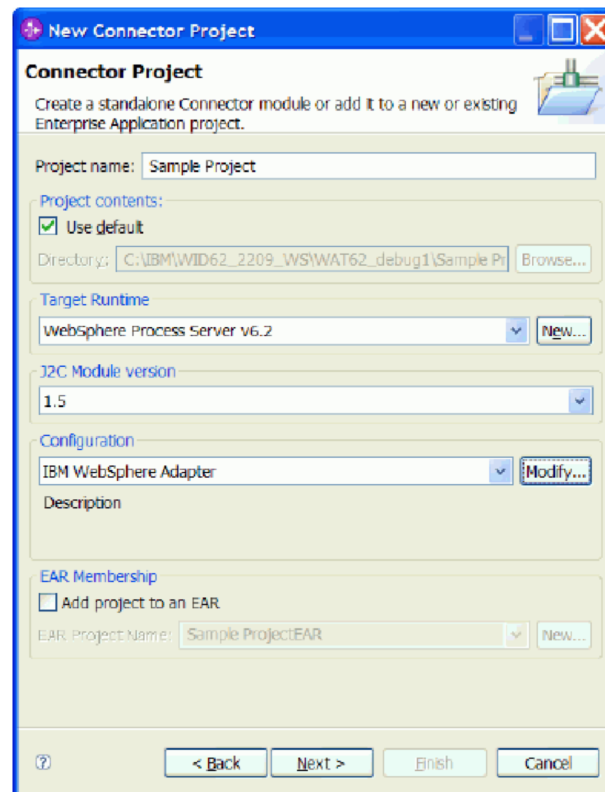
Connector Project dialog

You are ready to describe your project and resource adapter properties.

Specify project properties

You name your connector project, optionally adding it to an Enterprise Application project. You also specify the configuration used for developing the resource adapter.

You perform all of these tasks from the Connector Project dialog.



Connector Project dialog

1. Name your Connector Project.

   Enter a project name in the **Project name** field. The name is automatically appended to the workspace location.
2. Set **Target Runtime** to `<None>` and select `IBM Websphere Adapter` for the **Configurations**

   The `IBM Websphere Adapter` configuration contains the IBM WebSphere Adapter Foundation Classes 6.2, J2C Module 1.5 and Java 5.0 Project Facets. You can modify these facets by clicking the **Modify** button.
3. **Optional**: Specify an Enterprise Application Archive (EAR) project name.
   a. In the EAR Membership section, click the **Add project to an EAR** check box. Enter or select a name in the **EAR project Name** field or click **New** and then name the new EAR project.

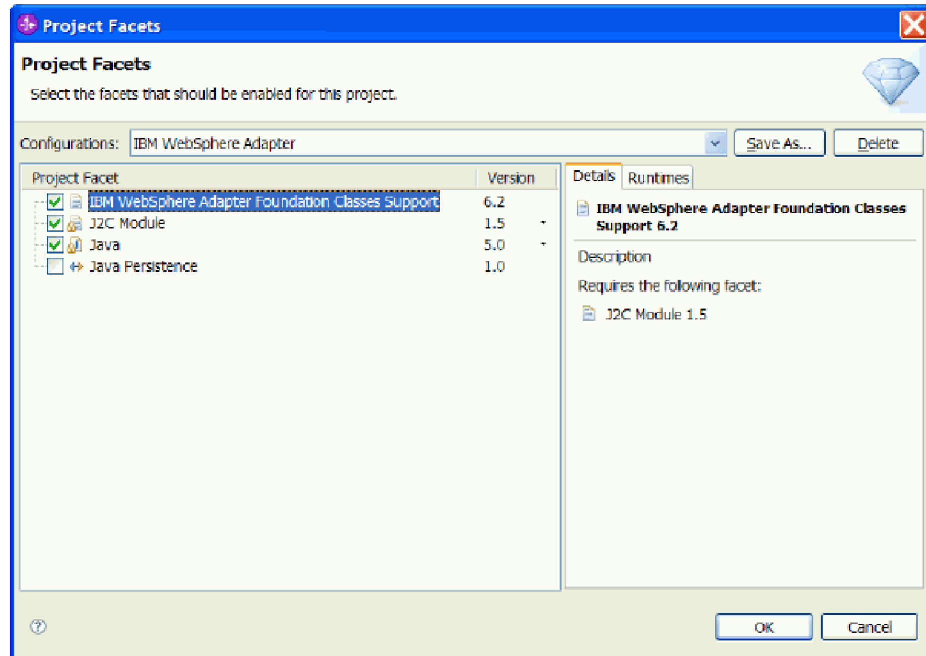
      Note: You can generate an EAR file later, after building and testing the adapter.
4. Click **Next**. This displays the Project facets dialog.

You are ready to specify project facet properties.

Specify project facets

As part of the process of creating a project, you specify *project facets*. A project facet represents a unit of functionality in the project. Project facets define characteristics and requirements for projects. When you add a facet to a project, that project is configured to perform a certain task, fulfill certain requirements, or have certain characteristics.

1. From the New Connector Project screen, make sure `IBM WebSphere Adapter` displays in the **Configurations** field and do not change the configuration or deselect any of the project facets. The project facets that are preselected are the facets you need to run the WebSphere Adapter Toolkit.



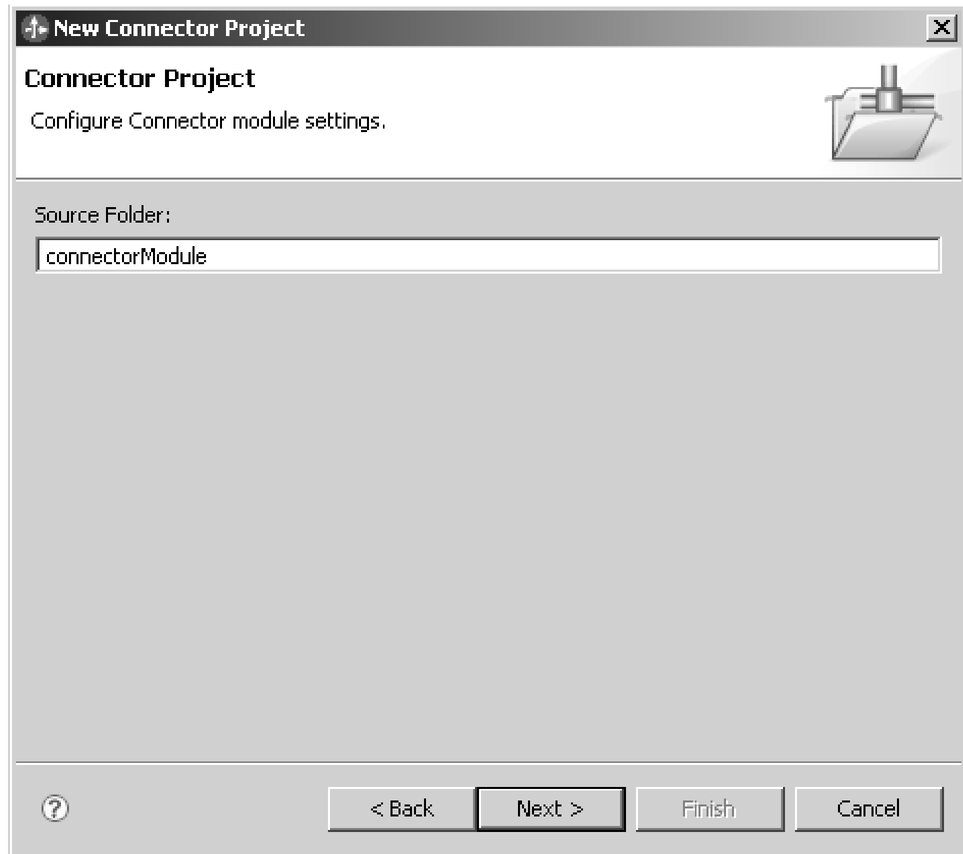
2. Select **Next** to go to the Connector Project module settings panel.

Now you can configure the Connector Project Module settings.

Specify connector project module settings

Specify connector project module settings for your adapter.

1. From the Configure Connector Module setting page, accept the name assigned to the connector project.



2. Click **Next** to advance to the Resource adapter properties page.

Now you are ready to set the properties for the resource adapter.

Specify resource adapter properties

Resource adapter properties are the descriptive properties that you assign to both the adapter and the adapter class.

You name the adapter and qualify its Java class with a package name and class prefix.

You perform these tasks in the J2C Resource Adapter Properties window.

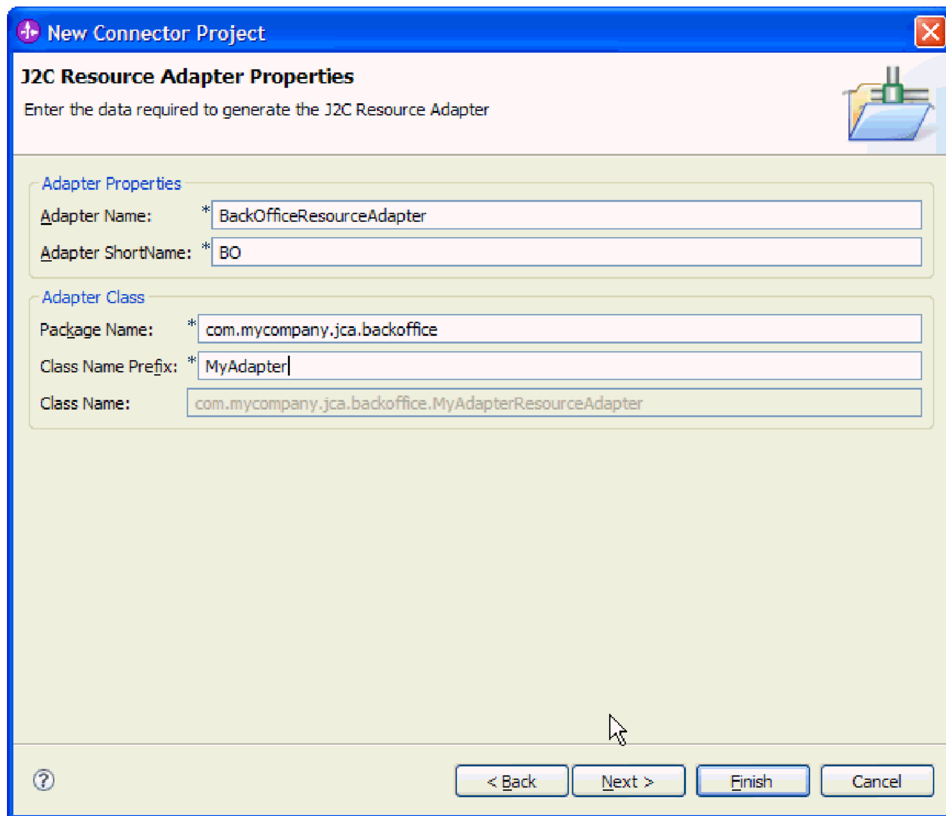


Figure 6. J2C Resource Adapter Properties window

1. In **Adapter Name**, type the name of the adapter.
2. In **Adapter ShortName**, type a one- to four-character short name for the adapter. The short name is used to create the component name that is used in the log and trace files as follows:

   ShortName + the characters "RA" + the value of the adapter ID property, which the user specifies when configuring the adapter

   For example, if you specify the short name `BO` and the user specifies the adapter ID of `001`, then the component name used in log and trace files is `BORA001`.
3. In **Package Name**, type the package name.
4. In **Class Name Prefix**, type the prefix to be used in adapter class names. **Class Name** displays the resulting fully qualified ResourceAdapter class name.
5. Click **Next**. This displays the Generation Options window.

You can now specify generation options for your resource adapter.

Specify generation options

Generation options are the types of components for which you can generate classes.

You specify adapter components in the Generation Options dialog.

You must choose which kind of adapter specification you want to implement: an IBM WebSphere Resource Adapter, or a J2C Resource Adapter. The adapter specification that you choose determines the generation options available.

For information on the characteristics of an IBM WebSphere Resource Adapter and a J2C Resource Adapter, see Introduction to JCA.

Generation Options dialog

From the **Adapter Specification** drop-down, choose the type of adapter you want to create:

- Choose **IBM WebSphere Resource Adapter** in the **Adapter Specification** list to generate code that extends the Adapter Foundation Classes.
- Choose **J2EE J2CA Resource Adapter** in the **Adapter Specification** list to generate code that implements the JCA 1.5 interface specification.

The wizard displays the generation options that correspond to your adapter.

What to do next

You are ready to specify generation options that match your adapter requirements.

Generating an IBM WebSphere Resource Adapter

You use the wizard to generate adapter classes that correspond to the adapter capabilities you require. This involves choosing the types of adapter classes (outbound, inbound, data binding, or enterprise metadata discovery) you want to generate and then choosing the component properties associated with the adapter classes.

The following sections describe the adapter classes and their associated component properties.

Outbound adapter classes and associated properties

Generating outbound adapter classes creates code (and in some cases sequence of calls, log and trace messages and comments) for the methods that must be implemented to produce a resource adapter that can send business events to an EIS. The list of Adapter Foundation Classes that are extended in your Connector Project when you choose to generate outbound adapter classes is as follows:

- **Connection** extends com.ibm.j2ca.base.WBIConnection
- **ConnectionFactory** extends com.ibm.j2ca.base.WBIConnectionFactory
- **ConnectionRequestInfo** extends com.ibm.j2ca.base.WBIConnectionRequestInfo
- **Interaction** extends com.ibm.j2ca.base.WBIInteraction
- **InteractionSpec** extends com.ibm.j2ca.base.WBIInteractionSpec
- **ManagedConnection** extends com.ibm.j2ca.base.WBIManagedConnection
- **ManagedConnectionFactory** extends com.ibm.j2ca.base.WBIManagedConnectionFactory

In addition the following are generated directly from the JCA 1.5 interface specification:

- **ConnectionSpec** implements javax.resource.cci.ConnectionSpec
- **ConnectionMetaDataImpl** implements javax.resource.cci.ConnectionMetaData
- **LocalTransaction** implements javax.resource.spi.LocalTransaction

You can select the following properties when generating outbound adapter classes:

- **Local transaction support**

  Generating outbound adapter classes with local transaction support means that the transaction is managed and performed by the EIS. *LocalTransaction* indicates the IBM WebSphere adapter supports local transactions. Local transaction support methods provide a LocalTransaction implementation and return the wrapper.

  Using the WBILocalTransaction covers the following requirements:

  – Notifies the JCA container

  – Updates the state of transactions on the WBIManagedConnection

  When you choose the `Local Transaction Support` component property from the list of outbound adapter component properties, the wizard will create the following method in the `ManagedConnection` class:

```
/**
* Does the EIS support local transaction?  Provide a LocalTransaction
* implementation and return the wrapper.
  *
  * @return  new instance of WBILocalResourceWrapper
  * @see  javax.resource.spi.ManagedConnection#getLocalTransaction()
 */
 public LocalTransaction getLocalTransaction() throws ResourceException {
   FooLocalTransaction transaction = new FooLocalTransaction(this);
     return new WBILocalTransactionWrapper(transaction, this);
 }
```

  For information on how to generate an outbound adapter with local transaction support, see Generating outbound local transaction support methods.

- **XA transaction support**

  Generating outbound adapter classes with XA transaction support means the transaction spans multiple heterogeneous systems. It uses global or two-phase-commit protocol. If a transaction manager coordinates a transaction, that transaction is considered a global transaction.

  Using the WBILocalTransaction covers the following requirements:

  – Notifies the JCA container

  – Updates the state of transactions on the WBIManagedConnection

  The getXAResource() method should get the XAResource from your EIS and return the wrapper. The WBIXAResourceWrapper acts as a thin layer delegating all calls to the underlying EIS XAResource instance, while at the same time tracking the sequence of calls to monitor and determine when the connection is involved in a transaction and when it is not.

  The wizard will create the `getXAResource()` and `getLocaltransaction()` method as follows:

  Local method

```
/**
* Does the EIS support local transaction?
*
* @return new instance of WBIXAResourceWrapper
* @see javax.resource.spi.ManagedConnection#getLocalTransaction()
*/
public LocalTransaction getLocalTransaction() throws ResourceException {
FooLocalTransaction transaction = new FooLocalTransaction(this);
return new WBILocalTransactionWrapper(transaction, this);
}
```

  XA method

```
/**
 * Does the EIS support XA transaction?
 * Get the XAResource from your EIS and return the wrapper.
 *
   * @return  new instance of WBIXAResourceWrapper
   * @see     javax.resource.spi.ManagedConnection#getXAResource()
  */
public XAResource getXAResource() throws ResourceException {
  return new WBIXAResourceWrapper(null, this);
}
```

Further implementation of these methods are needed to support XA transactions.

For information on how to generate XA Transaction support methods, see Generating XA Transaction support methods.

- **Command pattern**

  Generating command pattern classes allows you to break down a hierarchical update into a series of nodes and then generate a collection of sub-commands to manage the nodes. An interpreter processes the sub-commands, retrieving and executing the corresponding code. If you choose this option, the wizard generates code for the following classes:
  - **BaseCommand** extends com.ibm.j2ca.extension.commandpattern.CommandForCursor
  - **CommandFactory** implements com.ibm.j2ca.extension.commandpattern.CommandFactoryForCursor
  - **CreateCommand** extends <*your package name*>.outbound.commandpattern.<*your class prefix*>BaseCommand
  - **DeleteCommand** extends <*your package name*>.outbound.commandpattern.<*your class prefix*>BaseCommand
  - **NoOperationCommand** extends <*your package name*>.outbound.commandpattern.<*your class prefix*>BaseCommand
  - **RetrieveCommand** extends <*your package name*>.outbound.commandpattern.<*your class prefix*>BaseCommand
  - **RetrieveAllCommand** extends <*your package name*>.outbound.commandpattern.<*your class prefix*>BaseCommand
  - **UpdateCommand** extends <*your package name*>.outbound.commandpattern.<*your class prefix*>BaseCommand

  For information on how to generate command pattern classes, see Generating command pattern classes.

Inbound adapter classes and associated methods

Generating inbound adapter classes creates code for the methods that must be implemented to produce a resource adapter that can send events from an EIS to a business process. The list of Adapter Foundation Classes that are extended in your Connector Project when you choose to generate inbound adapter classes is as follows:

- **ActivationSpecWithXid** extends com.ibm.j2ca.base.WBIActivationSpecWithXid
- **EventStoreWithXid** extends com.ibm.j2ca.extension.eventmanagement.EventStoreWithXid

You can select the following properties when generating outbound adapter classes:

- **Connection pooling**

When you choose the connection pooling component property the wizard will create the `ActivationSpecWithXid` class that extends `WBIActivationSpecForPooling`.

For information on how to generate inbound connection pooling support, see Generating inbound connection pooling support.

- **Event polling support**

  Generating inbound adapter classes for event polling support creates code for the methods that must be implemented to produce a resource adapter that can send polling events from an EIS to a business process.

  The list of Adapter Foundation Classes that are extended in your Connector Project when you choose to generate inbound adapter classes for event polling support is as follows:

  – ActivationSpecWithXid extends com.ibm.j2ca.base.WBIActivationSpecWithXid

  – EventStoreWithXid extends com.ibm.j2ca.extension.eventmanagement.EventStoreWithXid

  When the CommException exception is logged during adapter startup because the EIS is down, stopped, or unreachable, the adapter automatically retries the connection if the RetryConnectionOnStartup activation specification property is enabled. If the property is not enabled, the adapter immediately reports the failure. This support is provided by the Adapter Foundation Classes. An adapter user enables this property in the external service wizard by selecting **Retry EIS connection on startup** on the Service Generation and Deployment Configuration window. The RetryConnectionOnStartup property works with the RetryLimit and RetryInterval properties, which specify the number of times the adapter retries the connection and the length of time it waits before retrying.

  For information on how to generate event polling support, see Generating inbound event polling support.

- **Generating inbound callback event support**

  Generating inbound adapter classes for callback event support creates code for the methods that must be implemented to produce a resource adapter that can send callback events from an EIS to a business process.

  The list of Adapter Foundation Classes that are extended in your Connector Project when you choose to generate inbound adapter classes for callback event support is as follows:

  – ActivationSpecWithXid extends com.ibm.j2ca.base.WBIActivationSpecWithXid

  – InboundListener that imports in com.ibm.j2ca.extension.eventmanagment.external.CallbackEventSender

  For information on how to generate inbound callback event support, see Generating inbound callback event support.

Data Binding classes

There are no properties associated with data binding classes.

Generating data binding classes creates code for the methods needed to marshall data from SDO to CCI record and from CCI record to SDO.

The list of Adapter Foundation Classes that are extended in your Connector Project when you choose to generate data binding classes is as follows:

- **DataBinding** implements

  `commonj.connector.runtime.RecordHolderDataBinding`

- **DataBindingGenerator** extends

`com.ibm.j2ca.extension.databinding.WBIDataBindingGenerator`

For information on how to generate data binding classes, see Generating Data Binding Classes.

Enterprise Metadata Discovery classes

There are no properties associated with Enterprise Metadata Discovery classes.

Generating enterprise metadata discovery classes creates code for the methods needed to produce a service that you can use to glean business object structure and other data from an EIS.

The list of Adapter Foundation Classes that are extended in your Connector Project when you choose to generate enterprise metadata discovery classes is as follows:

- **AdapterType** extends
  `com.ibm.j2ca.extension.emd.discovery.WBIAdapterTypeImpl`
- **DataDescription** extends
  `com.ibm.j2ca.extension.emd.description.WBIDataDescriptionImpl`
- **InboundConnectionConfiguration** extends
  `com.ibm.j2ca.extension.emd.discovery.connection.WBIInboundConnectionConfigurationImpl`
- **InboundConnectionType** extends
  `com.ibm.j2ca.extension.emd.discovery.connection.WBIInboundConnectionTypeImpl`
- **InboundServiceDescription** extends
  `com.ibm.j2ca.extension.emd.description.WBIInboundServiceDescriptionImpl`
- **MetadataDiscovery** extends
  `com.ibm.j2ca.extension.emd.discovery.WBIMetadataDiscoveryImpl`
- **MetadataEdit** extends
  `com.ibm.j2ca.extension.emd.discovery.WBIMetadataEditImpl`
- **MetadataImportConfiguration** extends
  `com.ibm.j2ca.extension.emd.discovery.WBIMetadataImportConfigurationImpl`
- **MetadataObject** extends
  `com.ibm.j2ca.extension.emd.discovery.WBIMetadataObjectImpl`
- **MetadataSelection** extends
  `com.ibm.j2ca.extension.emd.discovery.WBIMetadataSelectionImpl`
- **MetadataTree** extends
  `com.ibm.j2ca.extension.emd.discovery.WBIMetadataTreeImpl`
- **OutboundConnectionConfiguration** extends
  `com.ibm.j2ca.extension.emd.discovery.connection.WBIOutboundConnectionConfiguration`
- **OutboundConnectionType** extends
  `com.ibm.j2ca.extension.emd.discovery.connection.WBIOutboundConnectionTypeImpl`
- **OutboundServiceDescription** extends
  `com.ibm.j2ca.extension.emd.description.WBIOutboundServiceDescriptionImpl`

In addition to the Adapter Foundation Classes listed above, the following classes are also generated to assist you:

- **Constants**
  Contains enterprise metadata discovery constant variables.

- **StringCaseChanger**

  This is a utility that you can use format the business object or attribute name properly.

For information on how to generate Enterprise Metadata Discovery classes, see Generating Enterprise Metadata Discovery classes .

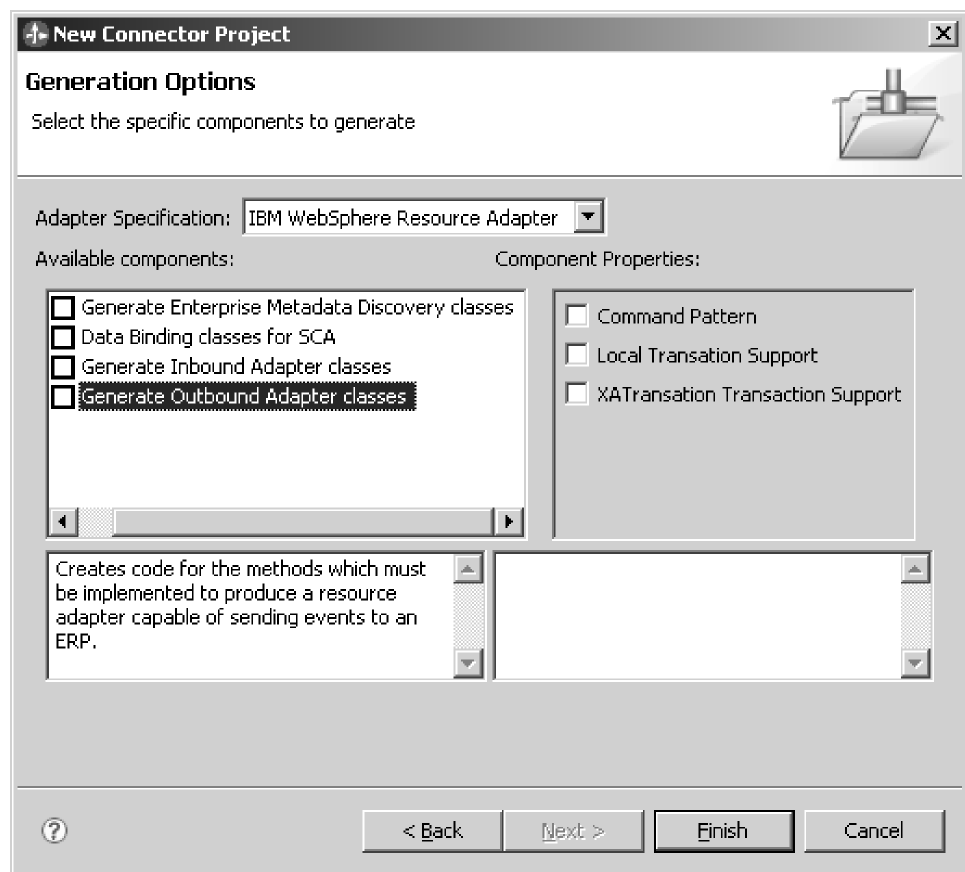
Generating outbound adapter classes

Generate outbound adapter classes for adapters that will send requests from a client application to the EIS.

Review the section on outbound adapter classes and associated properties in Generating an IBM WebSphere Resource Adapter.

When you choose to generate outbound adapter classes, the wizard creates code (and in some cases sequence of calls, log and trace messages and comments) for the methods that must be implemented to produce a resource adapter that can send business events to an EIS.

1. From the **Available components** portion of the Generation Options window, click the **Generate Outbound Adapter Classes** check box .



2. Review the available component property options associated with outbound adapter classes.

   Each of the component property options are described in the sections that follow.

Generate the outbound adapter classes for the component property selected.

Generating outbound local transaction support methods:

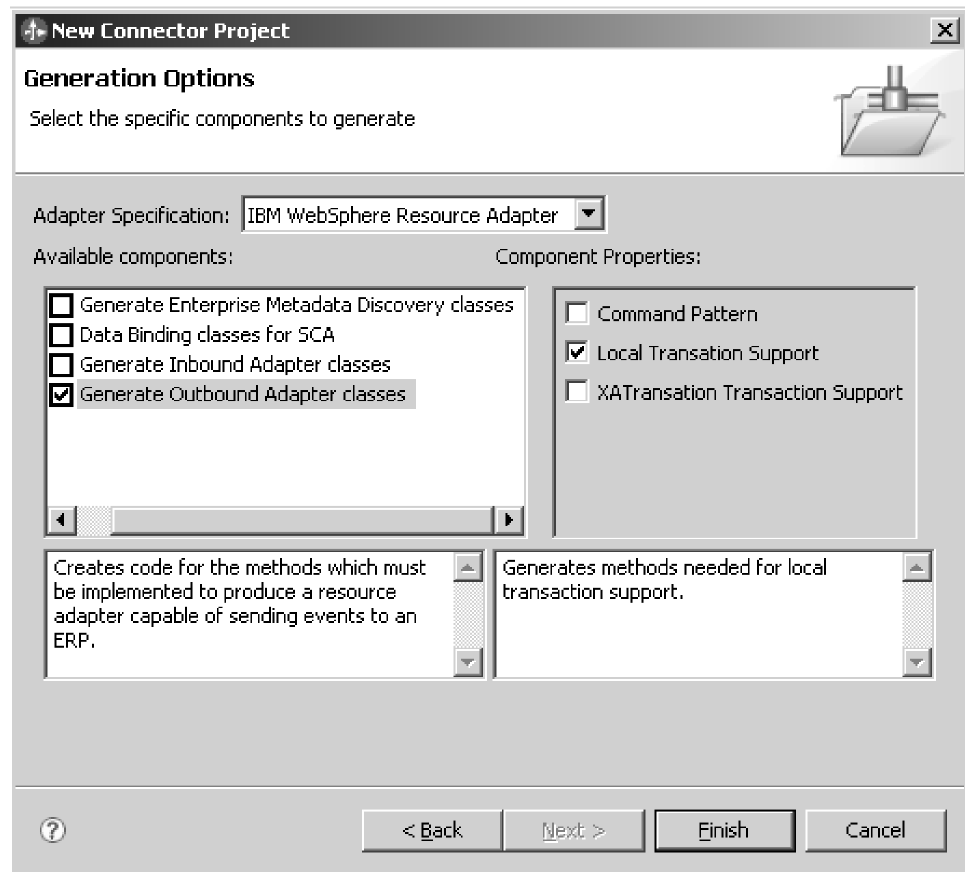
With local transaction support, the transaction is managed and performed by the EIS. *LocalTransaction* indicates the IBM WebSphere adapter supports local transactions. Local transaction support methods provide a LocalTransaction implementation and return the wrapper.

Consider your strategy for implementing transaction behavior in the adapter before generating outbound classes. You will not be allowed to regenerate code to add XA transactional behavior to the adapter if you click **Finish** to generate the code for this task.

Review the section on local transaction support in Generating an IBM WebSphere Resource Adapter.

For more information on transaction support, see Implementing transaction support.

1. Click the **Generate outbound adapter classes** check box and then click on the **Local Transaction Support** check box in the right pane.



Note: Before clicking **Finish** be aware that when you generate transactional support for the outbound classes you will not be able to modify the transaction support automatically after generating the classes.

2. Click **Finish**.

Now, you can generate outbound XA transaction support methods.

Generating outbound XA transaction support methods:

With XA transaction support, the transaction spans multiple heterogeneous systems. It uses global or two-phase-commit protocol. If a transaction manager coordinates a transaction, that transaction is considered a global transaction.

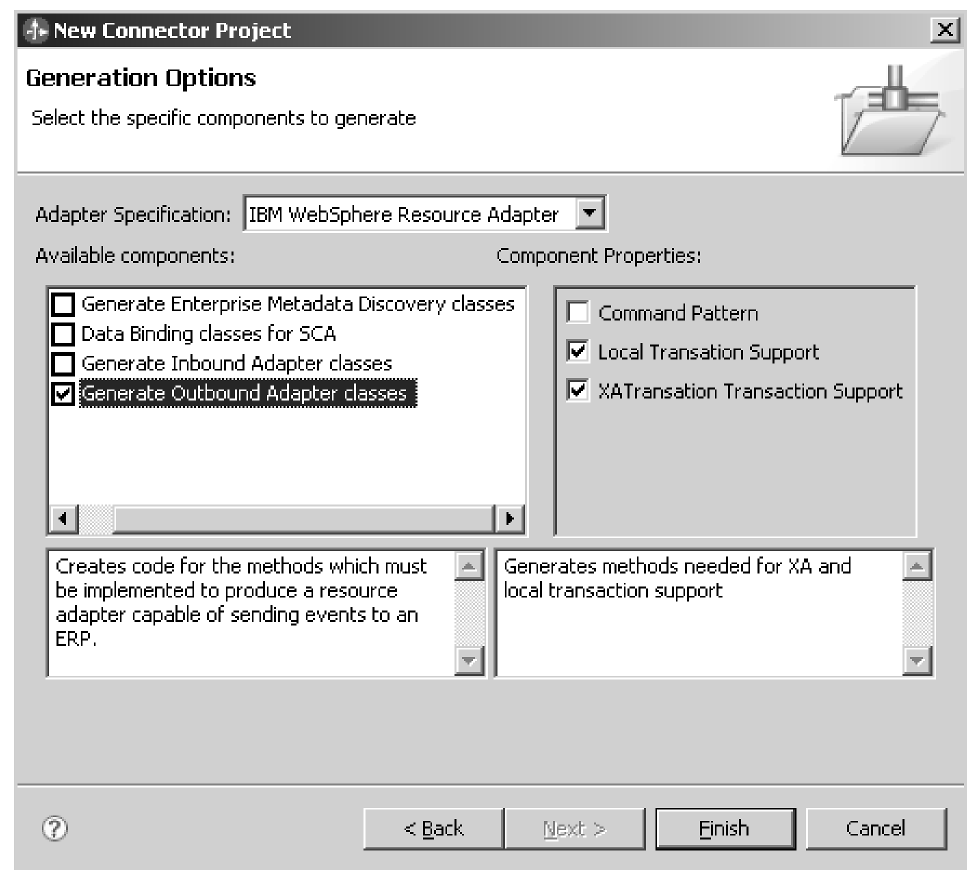
Review the section on XA transaction support in Generating an IBM WebSphere Resource Adapter.

The getLocalTransaction() method provides a LocalTransaction implementation and returns the wrapper.

For more information on transaction support, see Implementing transaction support.

1. Click the **Generate Outbound Adapter classes** check box and then click on the **XATransaction Transaction Support** check box in the right pane.

   When you choose **XATransaction Transaction Support**, **Local Transaction Support** is selected automatically.

2. Click **Finish**.

Now, you can generate command pattern classes.

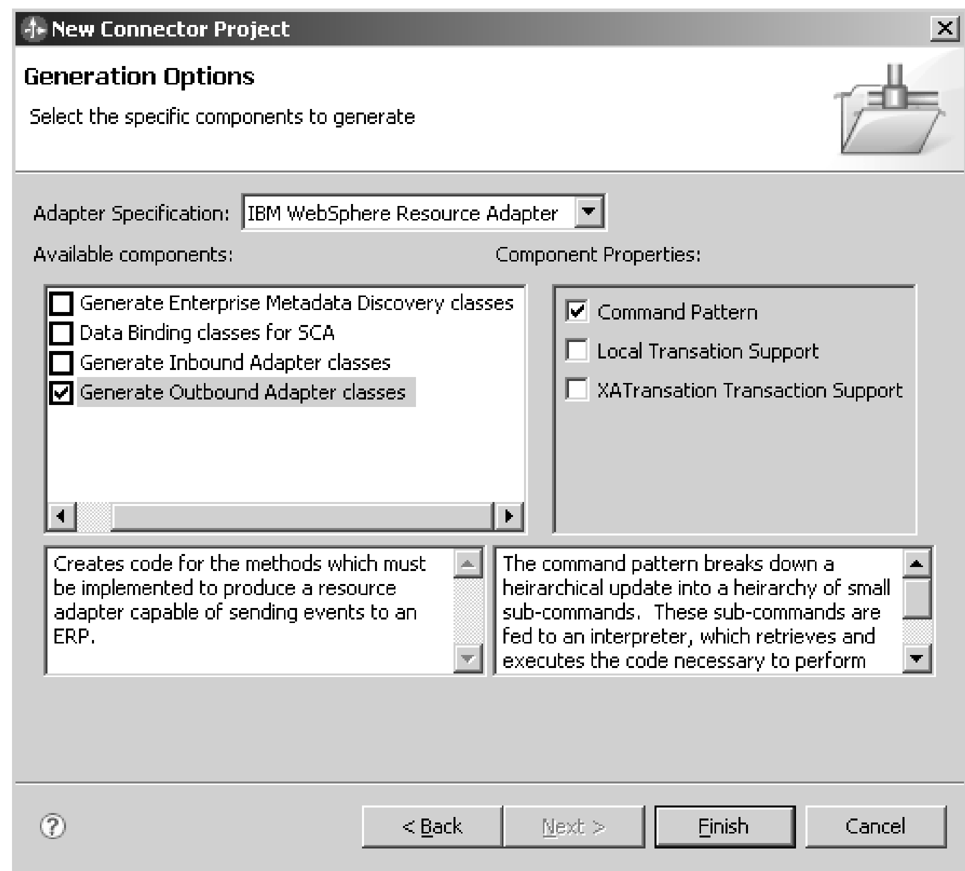
Generating command pattern classes:

Command pattern classes help reduce the complexity associated with comparing and managing dynamic object hierarchies.

Review the section on command pattern classes in Generating an IBM WebSphere Resource Adapter.

The command pattern classes allow you to break down a hierarchical update into a series of nodes and then generate a collection of sub-commands to manage the nodes. An interpreter processes the sub-commands, retrieving and executing the corresponding code.

1. Click **Generate Outbound Adapter Classes** and then click **Command Pattern** component property on the right pane.



2. Click **Finish**.

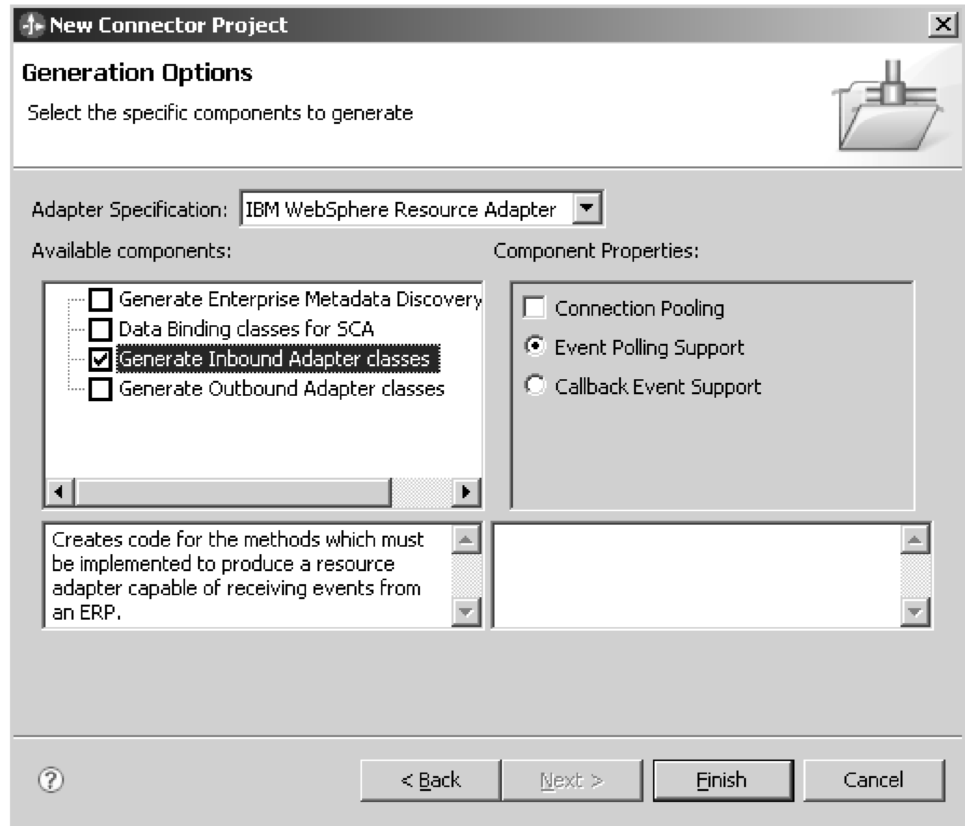
Generating inbound adapter classes

Generate inbound adapter classes for notifying a business process of an inbound event from the EIS.

Review the section on inbound adapter classes and associated properties in Generating an IBM WebSphere Resource Adapter.

When you choose to generate inbound adapter classes, the wizard creates code for the methods that must be implemented to produce a resource adapter that can send events from an EIS to a business process.

1. From the **Available components** portion of the Generation Options window, click the **Generate Inbound Adapter classes** check box.



2. Review the available component property options associated with inbound adapter classes.

   Each of the component property options are described in the sections that follow.

Generate the inbound adapter classes for the component property selected.

Note: Before deploying your adapter to WebSphere Process Server version 6.2, you must populate all inbound properties with default values. You can also assign dummy values to properties that are generated but not implemented by your adapter.

Generating inbound connection pooling support:

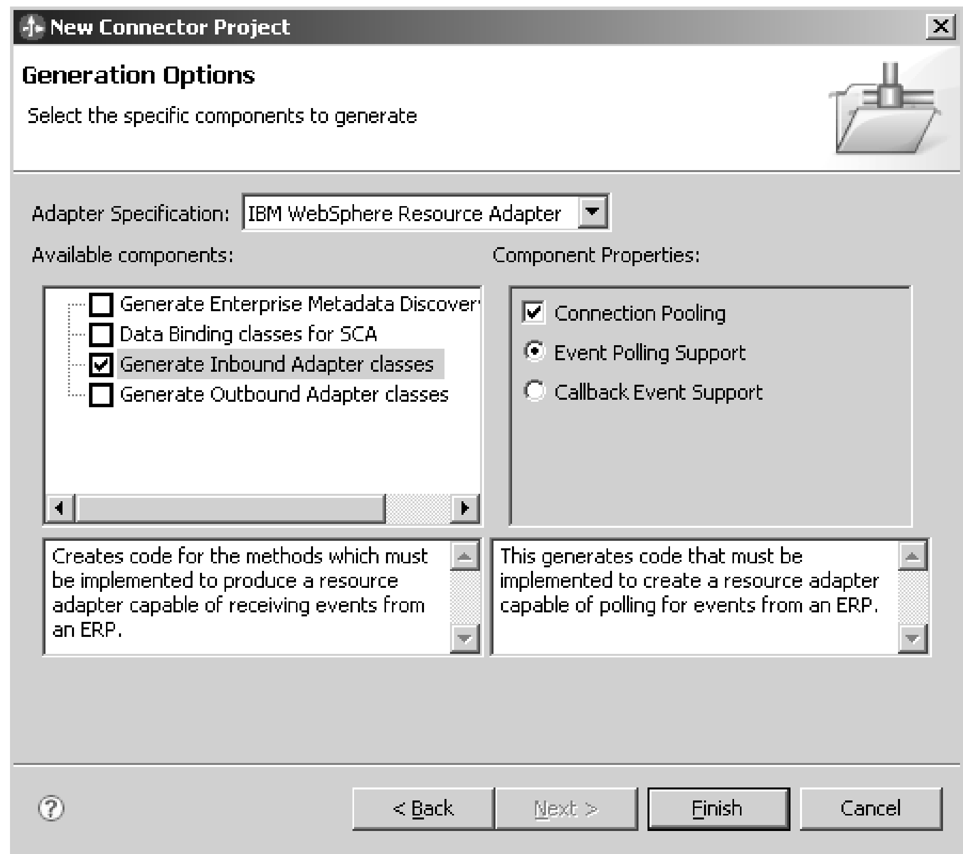
Generate inbound connection pooling support.

When you choose the connection pooling component property the wizard will create the `ActivationSpecWithXid` class that extends `WBIActivationSpecForPooling`.

Further implementation of this method is needed to support XA transactions.

1. Click the **Generate Inbound Adapter classes** check box and then click on the **Connection Pooling** check box in the right pane.

   Note: The **Event Polling Support** property is selected automatically when you select **Connection Pooling** . Connection pooling is not supported for Callback Events.



2. Click **Finish**.

Now, you can generate inbound event polling support.

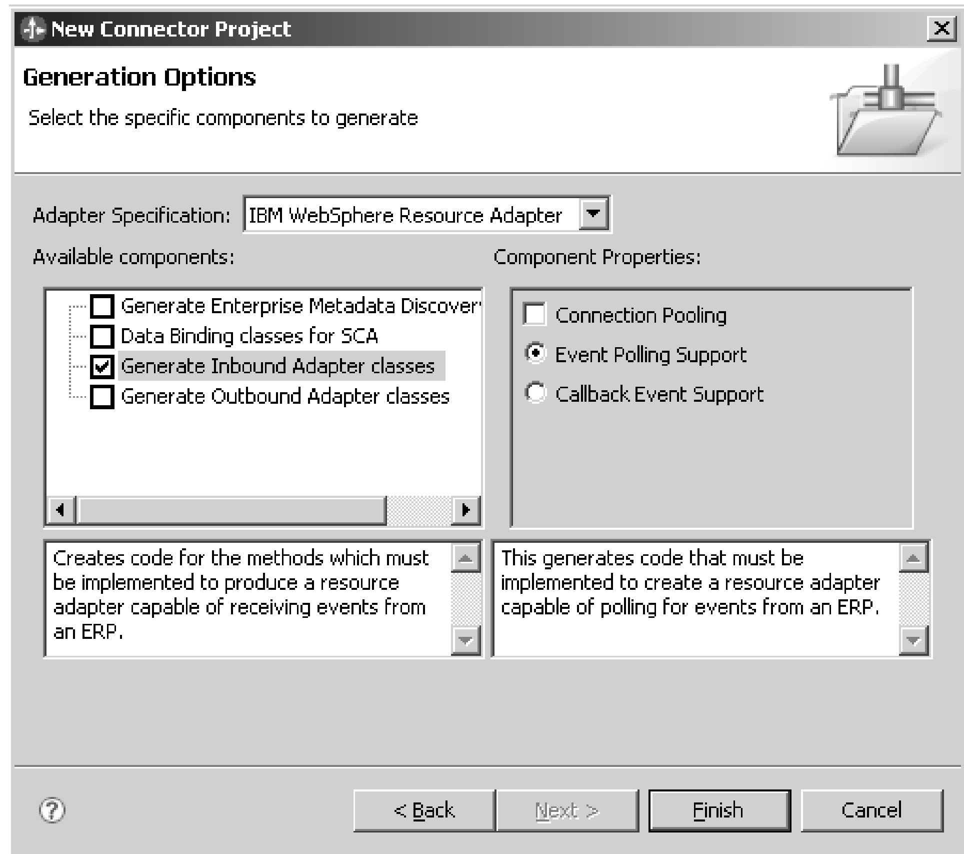
Generating inbound event polling support:

Event polling refers to an adapter's capability to poll the event records from the event store at regular intervals. In each poll call, a number of events are processed by the adapter.

Review the information on inbound adapter classes and the information on inbound event polling support in Generating an IBM WebSphere Resource Adapter.

When you choose to generate inbound adapter classes for event polling support, the wizard creates code for the methods that must be implemented to produce a resource adapter that can send polling events from an EIS to a business process.

1. Click the **Generate Inbound Adapter** check box and then click on the **Event Polling Support** check box in the right pane.



2. When you are finished choosing generation options, click **Finish**.

Now, you can generate inbound callback event classes.

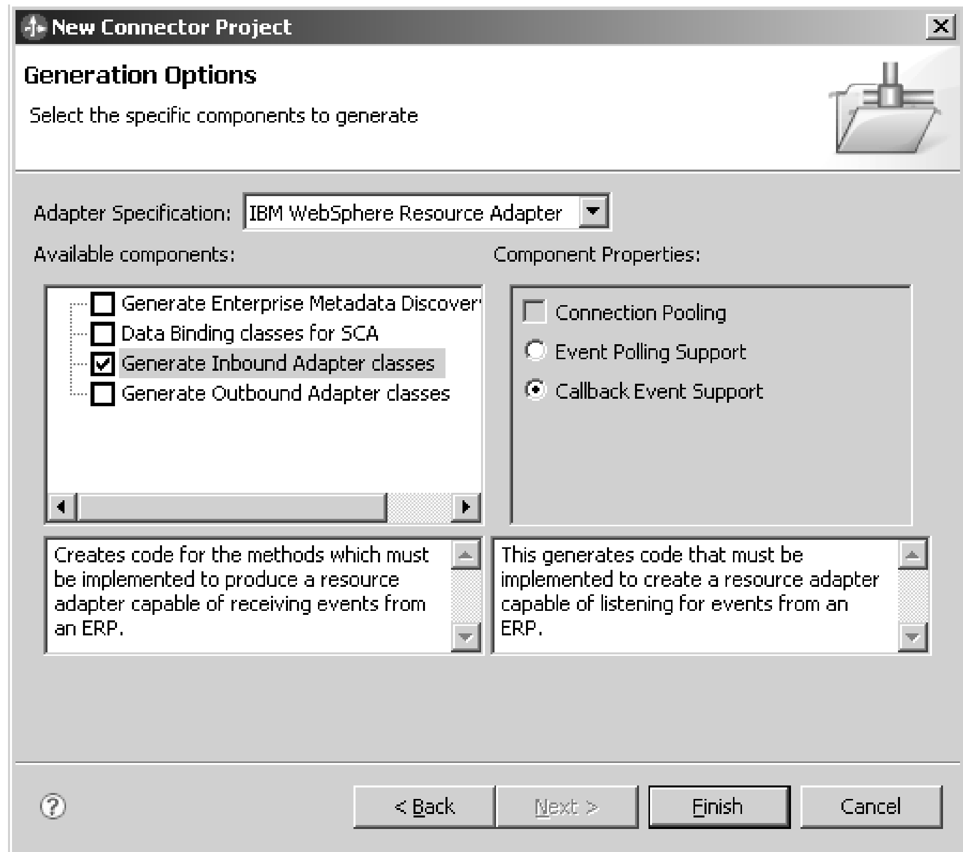
Generating inbound callback event support:

Inbound callback event support alerts business processes to changes in, or new information about, an EIS. The phrase callback refers to the ability of the EIS system to directly notify the adapter or business processes of a change, as opposed to the polling mechanism used in event notification.

Review the information on inbound adapter classes and the information on inbound callback event support in Inbound callback event notification.

When you choose to generate inbound adapter classes for callback event support, the wizard creates code for the methods that must be implemented to produce a resource adapter that can send callback events from an EIS to a business process.

1. Click the **Generate Inbound Adapter classes** check box and then click on the **Callback Event Support** check box in the right pane.



2. When you are finished choosing generation options, click **Finish**.

Now you can generate enterprise metadata discovery classes.

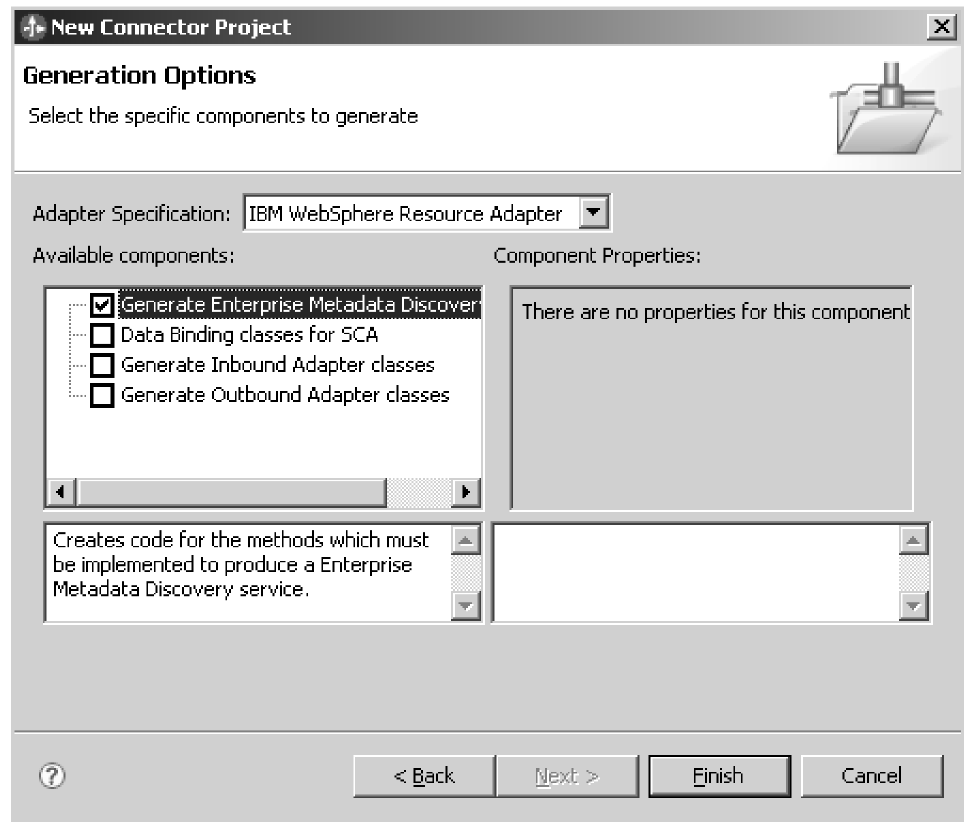
Generating enterprise metadata discovery classes

The enterprise metadata discovery classes are used by the external service discovery tool in WebSphere Integration Developer to introspect an EIS to create business objects and other artifacts.

Review the information on enterprise metadata discovery classes in Generating an IBM WebSphere Resource Adapter.

When you choose to generate enterprise metadata discovery classes, the wizard generates code for the methods needed to produce a service that you can use to glean business object structure and other data from an EIS.

1. Click the **Generate Enterprise Metadata Discovery classes** check box.



2. When you are finished choosing generation options, click **Finish**.

Generate data binding classes.

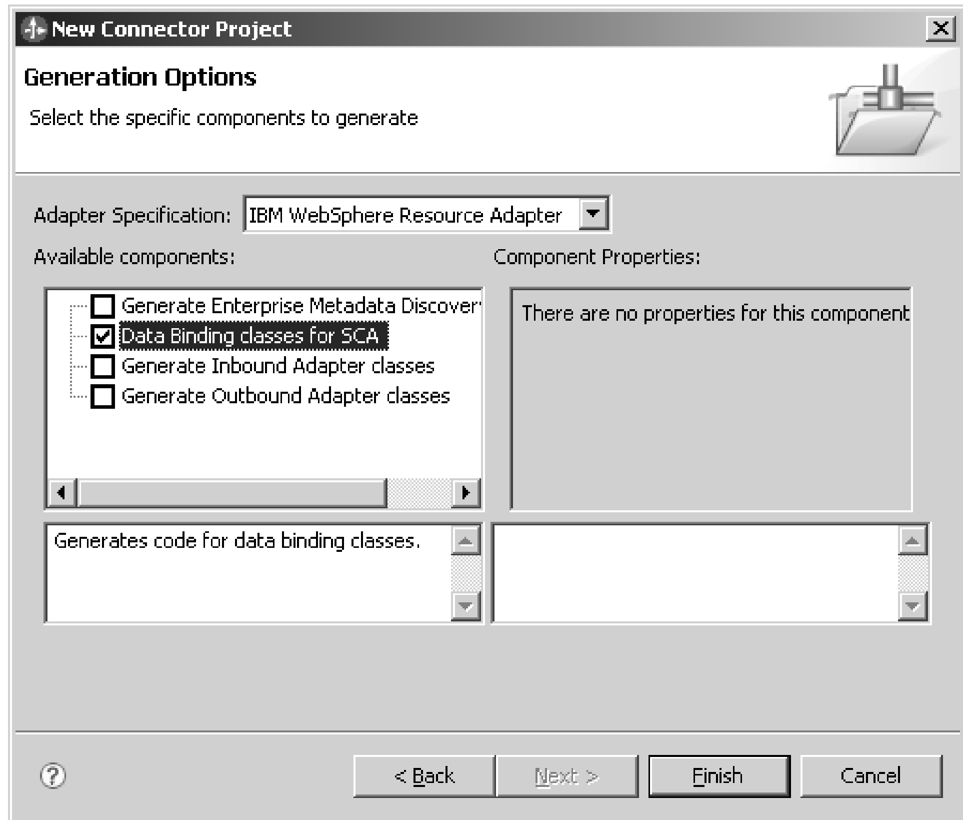
Generating data binding classes

You can generate data binding classes separate from the data binding classes that are generated from enterprise metadata discovery.

Review the section on data binding classes in Generating an IBM WebSphere Resource Adapter.

When you choose to generate data binding classes, the wizard generates code for the methods needed to marshall data from SDO to CCI record and from CCI record to SDO.

1. Click the **Generate Data Binding classes for SCA** check box.



2. When you are finished choosing generation options, click **Finish**.

Learn how to generate a JCA resource adapter.

Generating a JCA resource adapter

You use the wizard to generate adapter classes that correspond to the properties and options you specify.

The following sections describe the J2EE resource adapter classes.

Outbound JCA resource adapter classes

Generating outbound JCA resource adapter classes creates code for the methods that must be implemented to produce a JCA resource adapter that can send business events to an EIS. The list of JCA 1.5 interfaces that are implemented in your Connector Project when you choose to generate outbound adapter classes is as follows outbound JCA adapter classes:

- **ConnectionFactory** implements javax.resource.cci.ConnectionFactory
- **Connection** implements javax.resource.cci.Connection
- **ConnectionMetaData** implements javax.resource.cci.ConnectionMetaData
- **ConnectionRequestInfo** implements javax.resource.cci.ConnectionRequestInfo
- **ConnectionSpec** implements javax.resource.cci.ConnectionSpec
- **Interaction** implements javax.resource.cci.Interaction
- **InteractionSpec** implements javax.resource.cci.InteractionSpec
- **LocalTransaction** implements javax.resource.cci.LocalTransaction

- **ManagedConnectionFactory** implements javax.resource.spi.ResourceAdapterAssociation, javax.ValidatingManagedConnectionFactory, ManagedConnectionFactory
- **MangedConnection** implements javax.resource.spi.DissasociatableManagedConnection, ManagedConnection
- **ManagedConnectionMetaData** implements ManagedConnectionMetaData

For information on how to generate outbound JCA resource adapter classes, see Generating outbound JCA adapter classes.

Inbound JCA resource adapter classes

Generating inbound JCA resource adapter classes creates code for the methods that must be implemented to produce a resource adapter that can send events from an EIS to a business process. The list of JCA 1.5 interface classes that are implemented in your Connector Project when you choose to generate inbound adapter classes is as follows:

- **ActivationSpec** Implements javax.resource.spi.ActivationSpec

For information on how to generate inbound JCA resource adapter classes, see Generating inbound JCA adapter classes.

JCA Enterprise Metadata Discovery classes

Generating enterprise metadata discovery classes creates code for the methods needed to produce a service that you can use to glean business object structure and other data from an EIS. The wizard also generates a discovery-service.xml file.

The list of JCA interfaces that are implemented in your Connector Project when you choose to generate enterprise metadata discovery classes is as follows:

- **DataBindingDescription** implements commonj.connector.description.DataBindingDescription
- **DataBindingGenerator** implements commonj.connector.description.DataBindingGenerator
- **DataDescription** implements commonj.connector.description.DataDescription
- **DataFile** implements commonj.connector.description.DataFile
- **FunctionDescription** implements commonj.connector.description.FunctionDescription
- **InboundFunctionDescription** implements commonj.connector.description.InboundFunctionDescription
- **InboundServiceDescription** implements commonj.connector.description.InboundServiceDescription
- **OutboundFunctionDescription** implements commonj.connector.description.OutboundFunctionDescription
- **OutboundServiceDescription** implements commonj.connector.description.OutboundServiceDescription
- **SchemaDefinition** implements commonj.connector.SchemaDefinition
- **ServiceDescription** implements commonj.connector.description.ServiceDescription
- **AdapterType** implements commonj.connector.discovery.AdapterType
- **AdapterTypeSummary** implements commonj.connector.discovery.AdapterTypeSummary

- **EditableType** implements commonj.connector.discoveryEditableType
- **MetadataDiscovery** implements commonj.connector.discovery.MetadataDiscovery
- **MetadataEdit** implements commonj.connector.discovery.MetadataEdit
- **MetadataImportConfiguration** implements commonj.connector.discovery.MetadataImportConfiguration
- **MetadataObject** implements commonj.connector.discovery.MetadataObject
- **MetadataObjectIterator** implements commonj.connector.discovery.MetadataObjectIterator
- **MetadataObjectResponse** implements commonj.connector.discovery.MetadataObjectResponse
- **MetadataSelection** implements commonj.connector.discovery.MetadataSelection
- **MetadataTree** implements commonj.connector.discovery.MetadataTree
- **ConnectionConfiguration** implements commonj.connector.discovery.ConnectionConfiguration
- **ConnectionPersistence** implements commonj.connector.discovery.ConnectionPersistence
- **ConnectionType** implements commonj.connector.discovery.ConnectionType
- **InboundConnectionConfiguration** implements commonj.connector.discovery.InboundConnectionConfiguration
- **InboundConnectionType** implements javax.resource.emd.discovery.InboundConnectionType
- **MetadataConnection** implements commonj.connector.discovery.MetadataConnection
- **OutboundConnectionConfiguration** implements commonj.connector.discovery.OutboundConnectionConfiguration
- **OutboundConnectionType** implements commonj.connector.discovery.OutboundConnectionType
- **Action** implements commonj.connector.discovery.Action
- **ObjectWizard** implements commonj.connector.discovery.ObjectWizard
- **ObjectWizardStatus** implements commonj.connector.discovery.ObjectWizardStatus
- **ObjectWizardStep** implements commonj.connector.discovery.ObjectWizardStep
- **Operation** implements commonj.connector.discovery.Operation
- **OperationType** implements commonj.connector.discovery.OperationType
- **DataBinding** implements javax.resource.runtime.DataBinding
- **FunctionSelector** implements javax.resource.runtime.FunctionSelector
- **InboundListener** implements javax.resource.runtime.InboundListener
- **InboundNativeDataRecord** implements javax.resource.runtime.InboundNativeDataRecord
- **RecordDataBindingImpl** implements javax.resource.runtime.RecordDataBinding
- **RecordHolderDataBinding** implements javax.resource.runtime.RecordHolderDataBinding
- **PropertyDescriptor** implements javax.resource.runtime.PropertyDescriptor
- **PropertyType** implements javax.resource.runtime.PropertyType
- **Property** implements javax.resource.runtime.Property and javax.resource.runtime.PropertyDescriptor
- **SingleTypedProperty**

implements javax.resource.runtime.SingleTypedProperty and javax.resource.runtime.PropertyDescriptor

- **SingleValuedProperty** implements javax.resource.runtime.SingleValedProperty and javax.resource.runtime.PropertyDescriptor
- **PropertyGroup** implements javax.resource.runtime.PropertyGroup and javax.resource.runtime.PropertyDescriptor
- **MultiValuedProperty** implements javax.resource.runtime.MultiValuedPropertyProperty and javax.resource.runtime.PropertyDescriptor
- **BoundedMultiValuedProperty** implements javax.resource.runtime.BoundedMultiValuedProperty and javax.resource.runtime.PropertyDescriptor
- **NodeProperty** implements javax.resource.runtime.NodeProperty and javax.resource.runtime.PropertyDescriptor
- **TreeProperty** implements javax.resource.runtime.TreeProperty and javax.resource.runtime.PropertyDescriptor
- **TableProperty** implements javax.resource.runtime.TableProperty and javax.resource.runtime.PropertyDescriptor

For information on how to generate Enterprise Metadata Discovery classes, see Generating JCA Enterprise Metadata Discovery classes.

Note: If you select the IBM WebSphere Adapter Foundation Classes Support feature check box in the Project Facets screen, then the created J2EE resource adapter project automatically includes the IBM WebSphere Adapter Foundation Classes library in the project libraries list of the Java Build path.

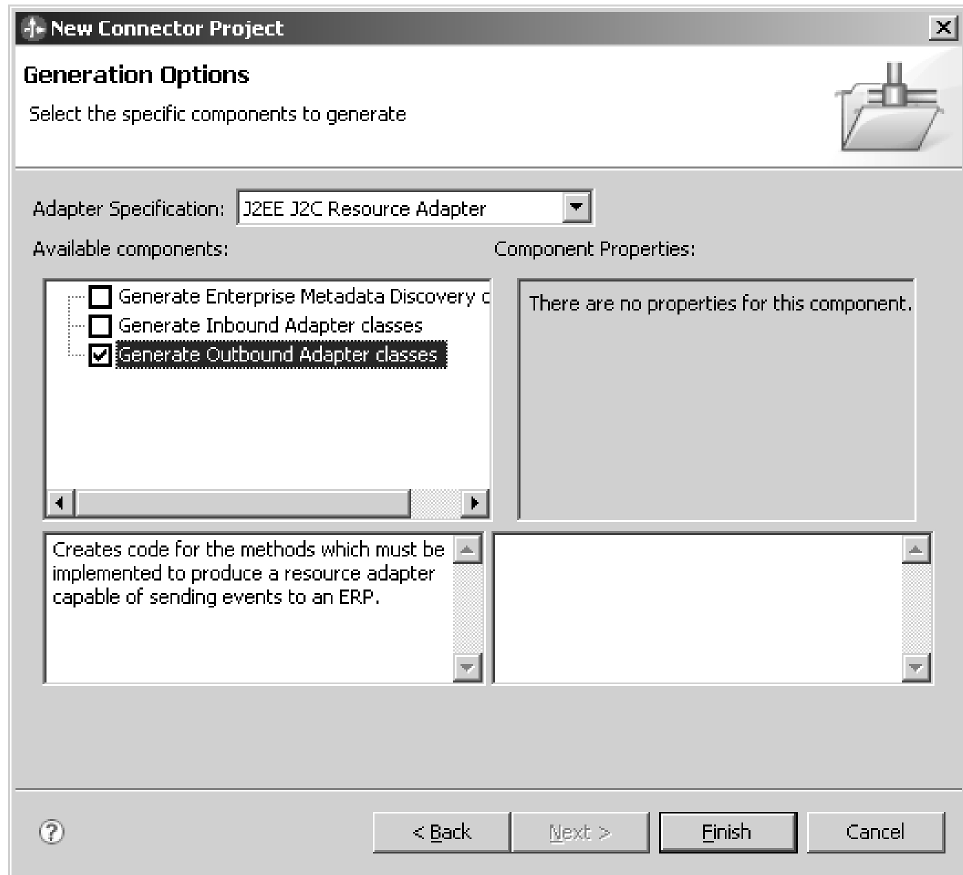
Generating outbound JCA adapter classes

The outbound adapter classes are responsible for notifying an EIS of outbound events from a business process.

Review the section on outbound JCA resource adapter classes in Generating a JCA resource adapter.

When you choose to generate outbound adapter classes, the wizard creates code for the methods that must be implemented to produce a resource adapter that can send business events to an EIS.

1. Click the **Generate Outbound Adapter Classes** check box.



2. When you are finished choosing generation options, click **Finish**.

Generate inbound JCA adapter classes.

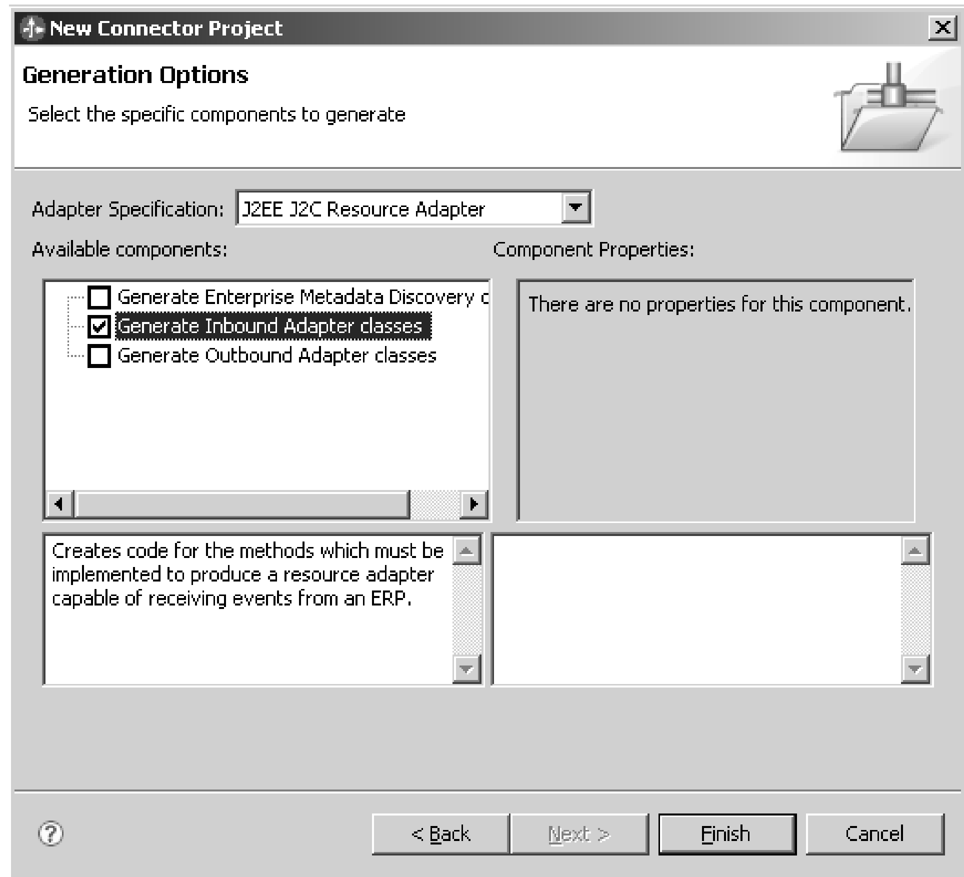
Generating inbound JCA adapter classes

The inbound adapter classes are responsible for notifying a business process of an inbound event from the EIS.

Review the section on inbound JCA resource adapter classes in Generating a JCA resource adapter.

When you choose to generate inbound adapter classes, the wizard creates code for the methods that must be implemented to produce a resource adapter that can send events from an EIS to a business process.

1. Click the **Generate Inbound Adapter classes** check box.



2. When you are finished choosing generation options, click **Finish**.

Generate JCA enterprise metadata discovery classes.

Generating JCA enterprise metadata discovery classes

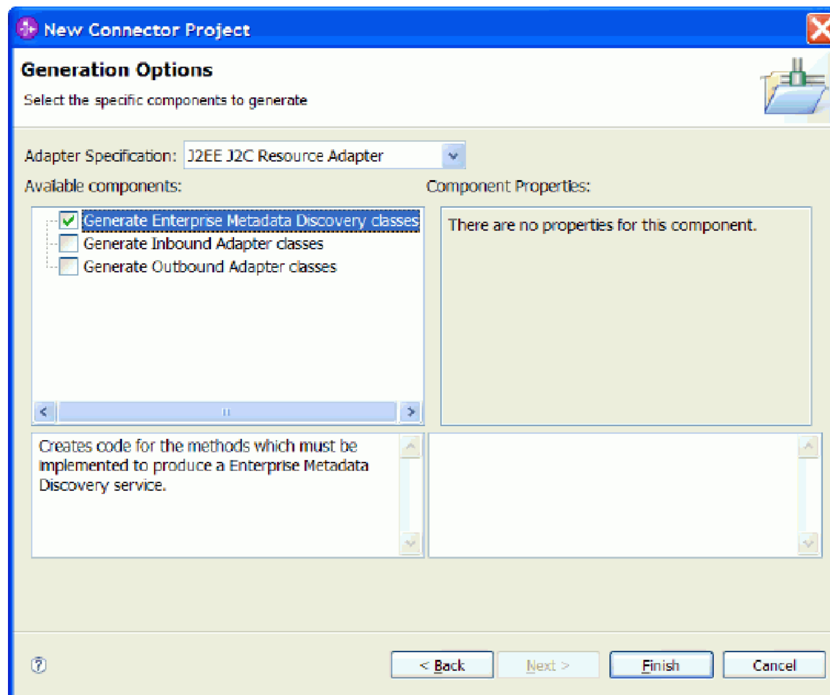
The enterprise metadata discovery classes are used by the external service discovery tool in WebSphere Integration Developer to introspect an EIS to create business objects and other artifacts.

Review the section on JCA enterprise metadata discovery classes in Generating a JCA resource adapter

When you choose to generate enterprise metadata discovery classes, the wizard generates code for the methods needed to produce a service that you can use to glean business object structure and other data from an EIS. The wizard also generates a discovery-service.xml file.

Note: When you generate JCA enterprise metadata discovery classes, the wizard adds a prefix to each. The prefix is the **Class Name Prefix** that you entered when specifying properties on the Resource adapter properties page of the wizard.

1. Click the **Generate Enterprise Metadata Discovery classes** check box.



2. When you are finished choosing generation options, click **Finish**.

Generated code and deployment descriptor

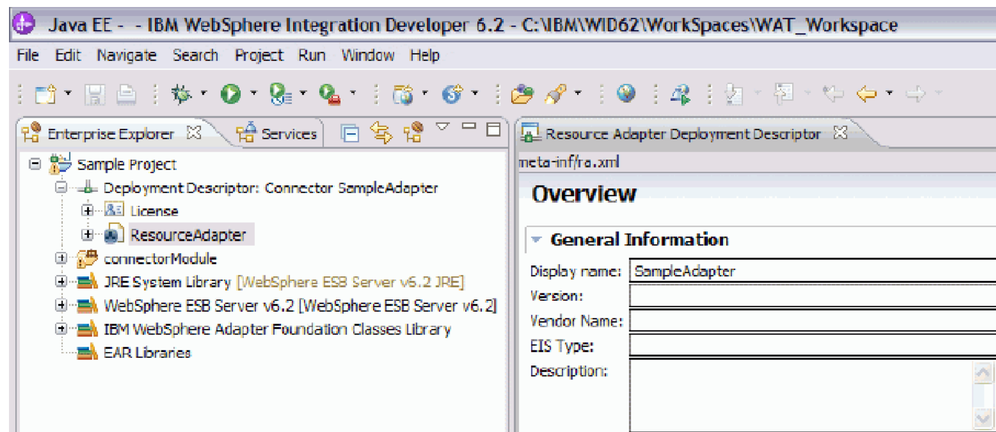
The generated artifacts reflect the adapter classes with the properties and options you specified.

After you specify options for your resource adapter, the wizard generates code and a deployment descriptor in a Connector Project and then switches to the J2EE perspective in the workspace. The wizard then automatically launches the Resource Adapter Deployment Descriptor editor.

For a complete description of the editor, see the “Resource Adapter Deployment Descriptor Editor overview” on page 10.

Using the Resource Adapter Deployment Descriptor editor

The Resource Adapter Deployment Descriptor editor provides an easy and convenient way to configure your resource adapter.



Resource Adapter Deployment Descriptor editor

You configure the resource adapter by using a deployment descriptor. The deployment descriptor—the ra.xml file— is generated by the wizard and included in your Java Connector Project.

The editor is made up multiple pages each of which represents a major section of the ra.xml file. Changes made in the editor are saved directly to the ra.xml file in your Java Connector Project.

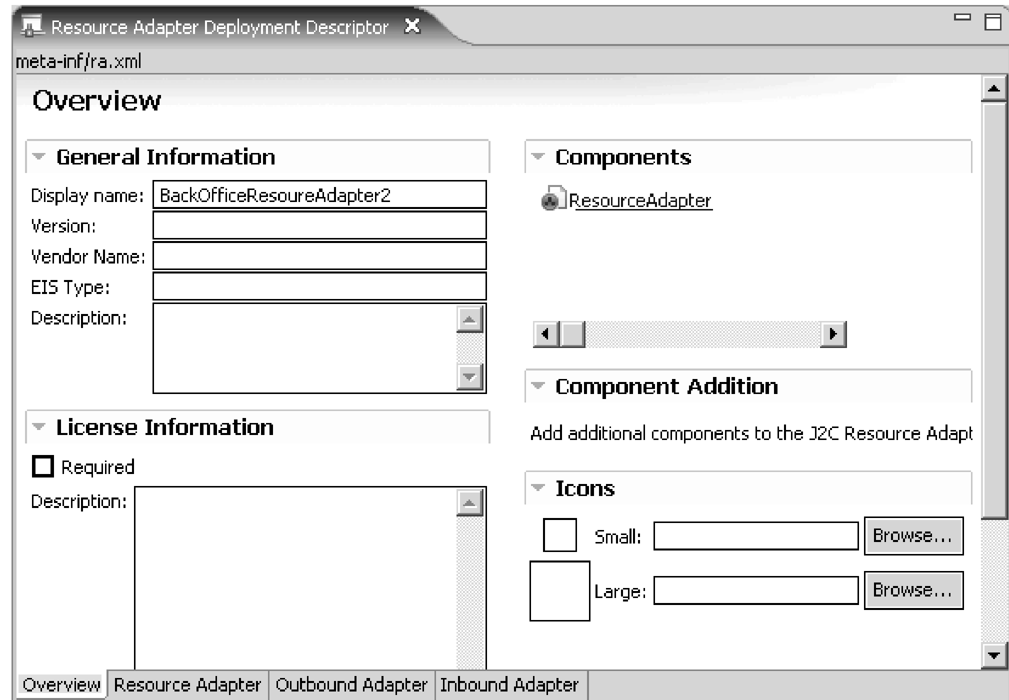
The editor allows you to display and modify all elements of the ra.xml file. You can also add properties. After each modification this file is validated against its schema definition to ensure its validity.

Any validation problems with the ra.xml file are displayed in the **Problems** view of the development environment.

Displaying the deployment descriptor

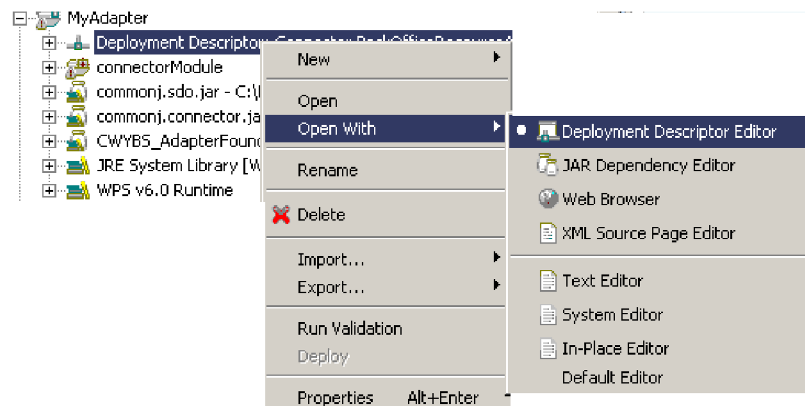
Once you complete the New Connector Project wizard your workspace is switched to the J2EE perspective and your deployment descriptor displays using the Resource Adapter Deployment Descriptor editor.

Displaying the deployment descriptor in the editor provides you with several different views.



Deployment descriptor Overview pane

Alternatively, you can view the deployment descriptor in the editor by highlighting the file in the Project Explorer and selecting **Open With** → **Deployment Descriptor Editor** from the context menu.



Displaying the descriptor editor from the context menu

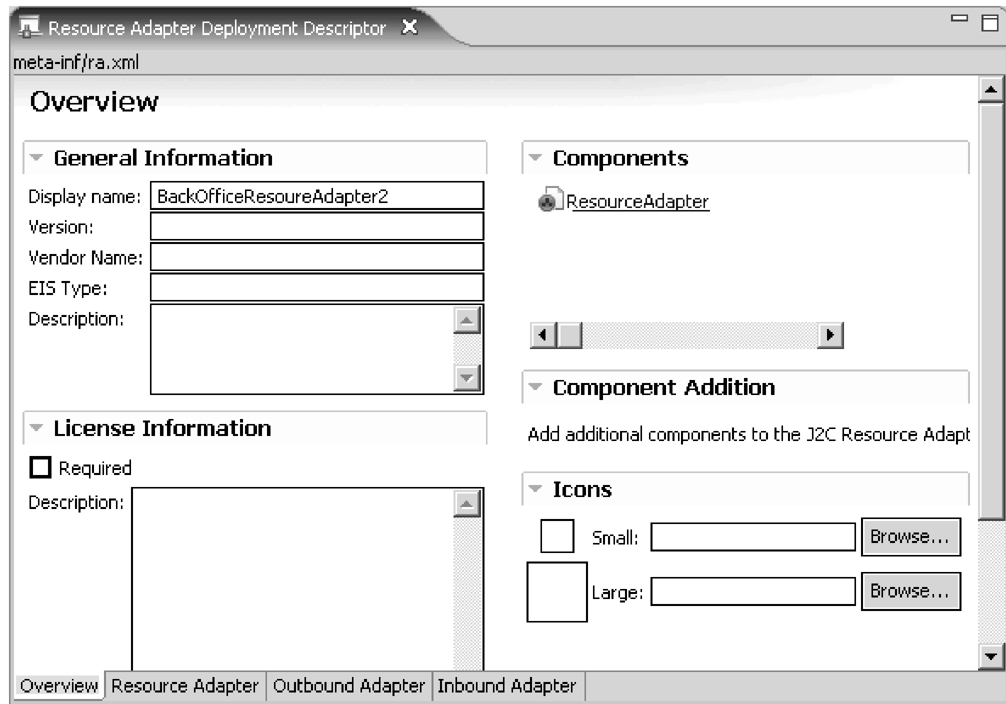
You can display each of the four views by using the tabs at the bottom of the Overview pane, which is the default view.

Descriptor editor tabs

Using the Overview pane

The Overview pane provides access to general information about your resource adapter. You can display it at any time by clicking the **Overview** tab at the bottom of this pane.

In addition to providing general information about your resource adapter, you can generate components that you may not have originally specified when working with the wizard.



Overview pane

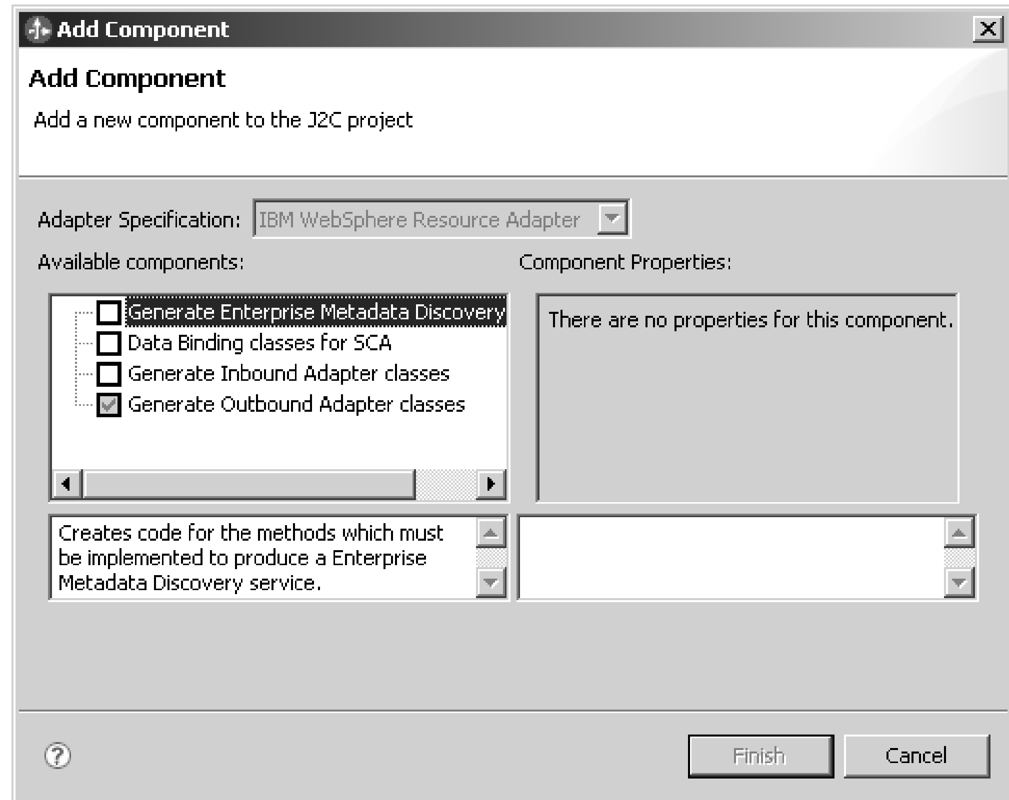
The **General Information** section summarizes general information about the resource adapter.

The **License Information** section allows you to specify and display license descriptions and requirements, if any, for the resource adapter.

The **Components** section provides navigation links to other panes of the editor.

The **Component Addition** section allows you to generate components that you may not have originally specified when working with the wizard. Click the **Add** to display the **Add Component** dialog box. Only those options not previously generated will be enabled for you to select.

Note: To display the **Add** button for **Add Component**, maximize the Overview pane. The **Add** button does not display when this pane is minimized.



Add component dialog

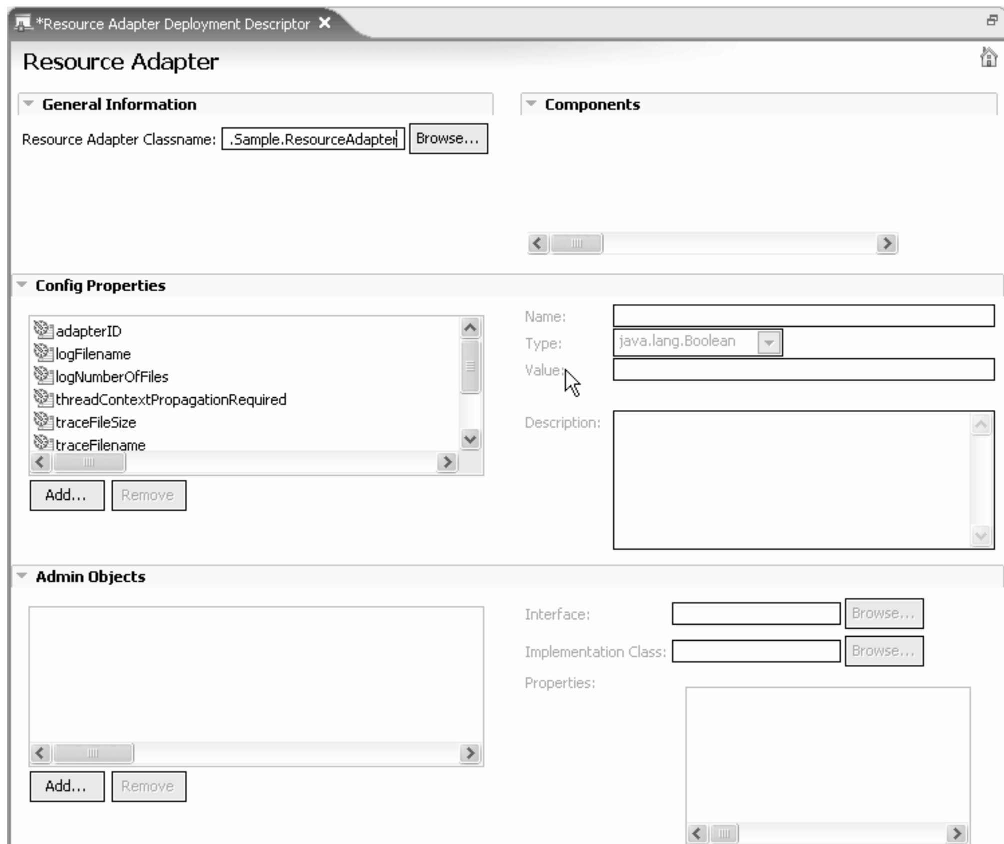
The **Icons** section of the overview pane allows you to associate icons with the resource adapter. You can specify a large or small icon in jpg or gif format. To fit into the allotted area, the large icons must be 32 x 32 pixels and the small icon 16 x 16 pixels. WebSphere Process Server does not make use of these icons.

Using the Resource Adapter pane

The Resource Adapter pane displays information that is stored in the `<resourceadapter>` element of the ra.xml file.

You can navigate to this pane by clicking **ResourceAdapter** in the Components section of the Overview pane or by clicking the **ResourceAdapter** tab at the bottom of the pane.

The sections of the Resource Adapter pane are described below.

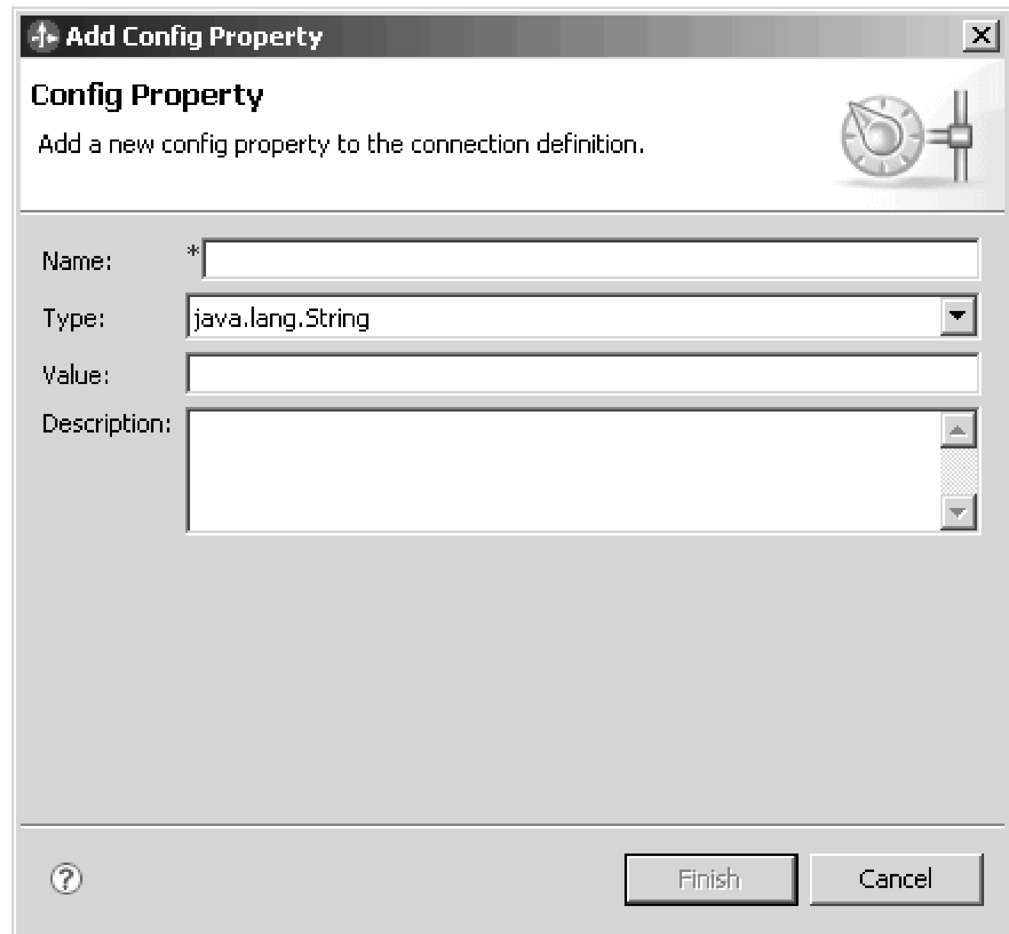


Resource adapter pane

The **General Information** section allows you to specify deployment descriptor values for the entire resource adapter. This section displays the name of the ResourceAdapter class with which this deployment descriptor is associated. This class must directly or indirectly implement the javax.resource.spi.ResourceAdapter interface.

The **Components** section provides navigation links to other parts of the editor. The Resource Adapter pane displays links to the Outbound Adapter and Inbound Adapter panes.

The **Config Properties** section allows you to specify resource-adapter-level configuration properties. By default, all configuration properties inherited by the class specified in the Resource Adapter Classname field in the General Information section are shown here. You can specify default values for these inherited properties. In addition, you can add or delete your own user-defined configuration properties using the **Add** and **Remove** buttons on the left side of the editor. You can also edit these properties using the widgets on the right side of the editor.



Add Config property dialog

When you add, modify, or delete user-defined properties in this section, the editor creates (or removes) the corresponding Java bean properties in your code. For more information, see Modifying properties.

The **Admin Objects** section allows you to specify administered objects for this resource adapter and configure their properties. You can add, update, or delete administered objects and specify properties for each object. The information you specify here will be stored at the resource adapter level of the deployment descriptor.

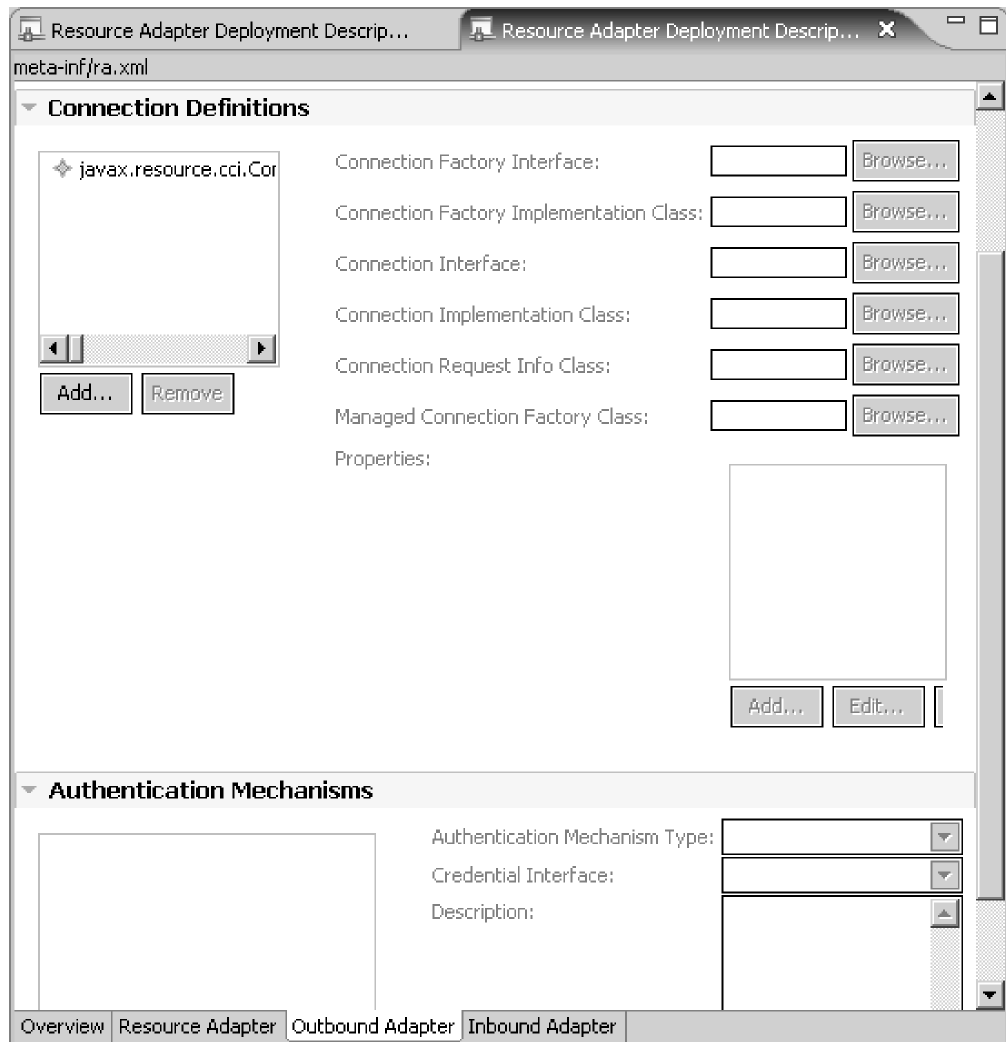
The **Security Permissions** section allows you to specify application server permissions. These are the permissions required by the resource adapter to execute within the application server. You can add, update, or delete security permissions. The information you specify here will be stored at the resource adapter level of the deployment descriptor.

Using the Outbound Adapter pane

The Outbound Adapter pane displays information that is stored in the `<outboundresourceadapter>` element of the ra.xml.

You can navigate to this pane by clicking **OutboundResourceAdapter** in the Components section of the Overview pane or by clicking the **Outbound Adapter** tab at the bottom of the pane.

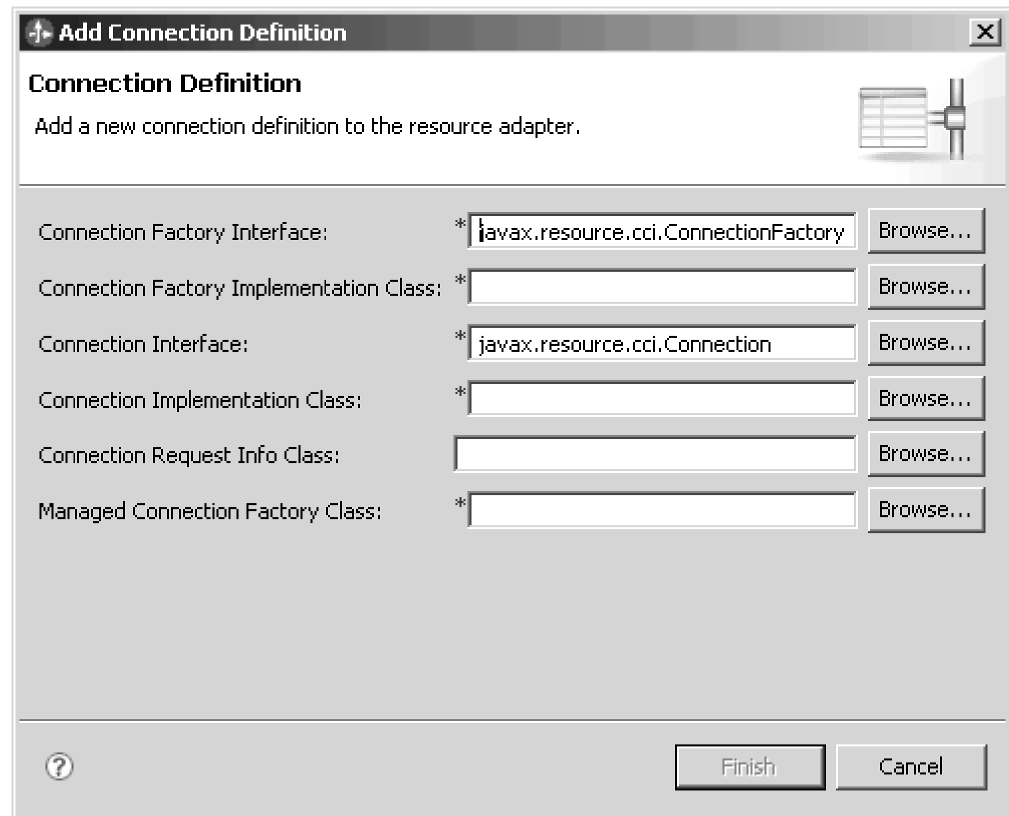
The sections of the Outbound Adapter pane are described below.



Outbound resource adapter pane

The **General Information** section allows you to specify deployment descriptor values associated with the outbound resource adapter. You can specify the level of transaction support, if any, and whether reauthentication is supported.

The **Connection Definitions** section allows you to specify an outbound connection definition. This connection definition is used by the application server to obtain a physical connection to the EIS. Clicking the **Add** button under the **Connection Definitions** list on the left side of the editor displays the following dialog box, which allows you to specify all required connection definition information.

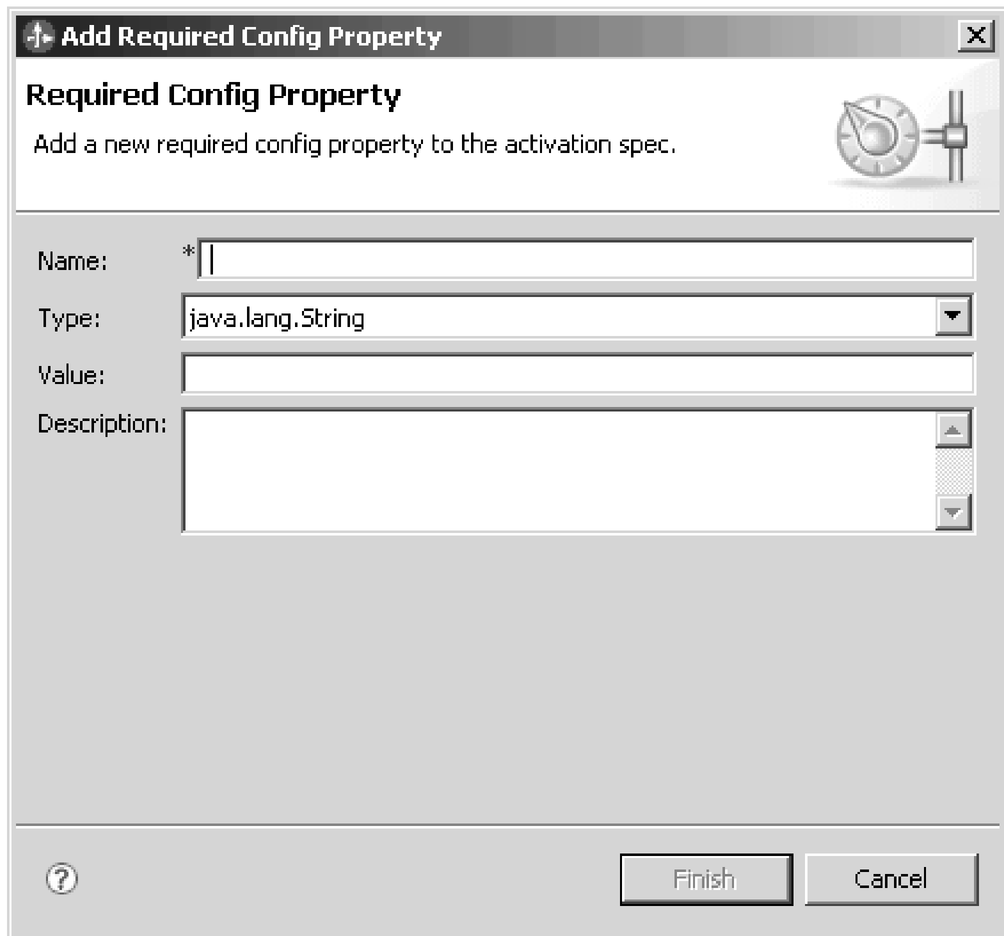


Add Connection Definition dialog

A **Connection Definition** requires the following information:

- ConnectionFactory Interface
- ConnecitonFactory Implementaion Class
- Connection Interface
- Connection Implementation Class
- ConnectionRequestInfo Class
- ManagedConnectionFactory Class

Once a connection definition is defined all properties inherited by the specified ManangedConnectionFactory are shown in the properties list directly under the ManagedConnectionFactory Class. You can specify default values and descriptions for these inherited properties. In addition, you can add, update or delete your own user-defined configuration properties using the **Add**, **Edit**, and **Remove** buttons on the right side of the editor.



Add Config property dialog

When you add, modify, or delete user-defined properties in this section, the editor creates (or removes) the corresponding Java bean properties in your code. For more information, see Modifying properties.

The **Authentication Mechanisms** section allows you to specify the authentication mechanism(s) supported by the resource adapter.

Using the Inbound Adapter pane

The Inbound Adapter pane displays information that is stored in the `<inbound-resourceadapter>` element of the ra.xml file.

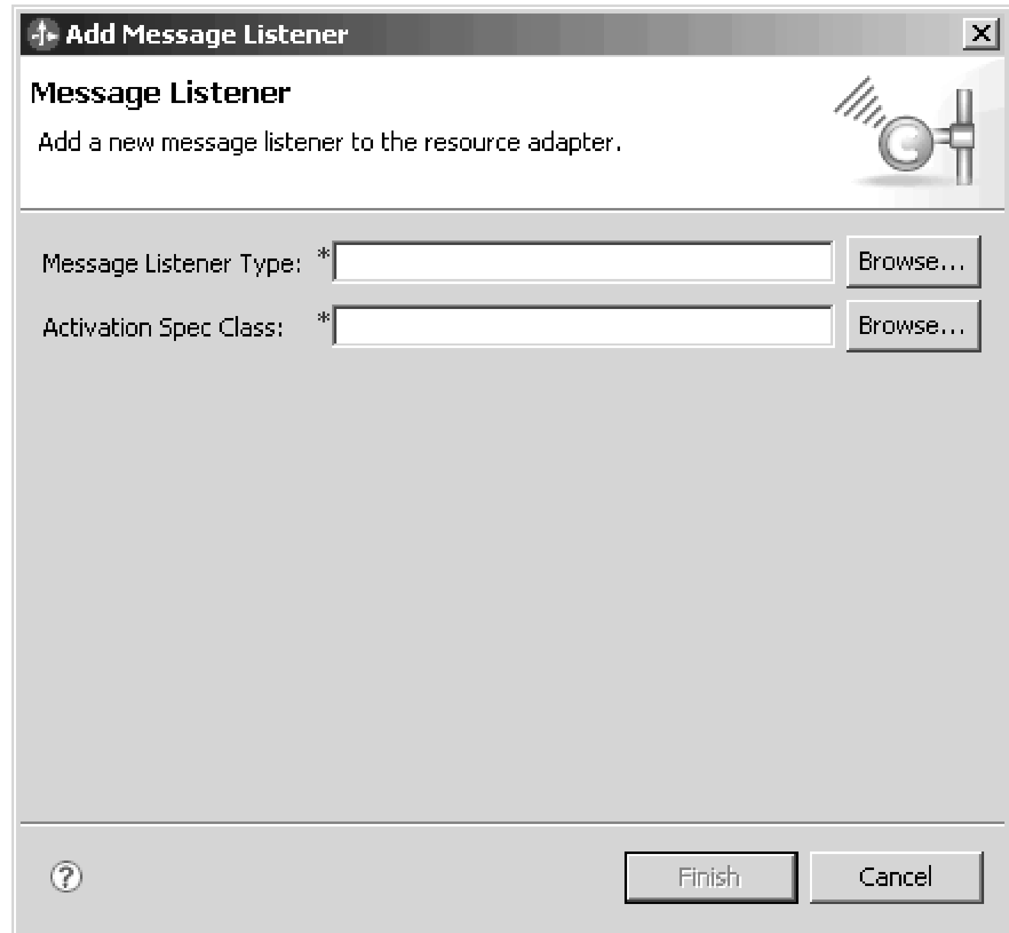
You can navigate to this pane by clicking **InboundResourceAdapter** in the Components section of the Overview pane or by clicking the **Inbound Adapter** tab at the bottom of the pane.

The Inbound Adapter pane contains a **Message Listeners** section.

Inbound Resource Adapter pane

The **Message Listeners** section allows you to specify message listeners for inbound event processing. You must specify a MessageListener type and an Activation Spec

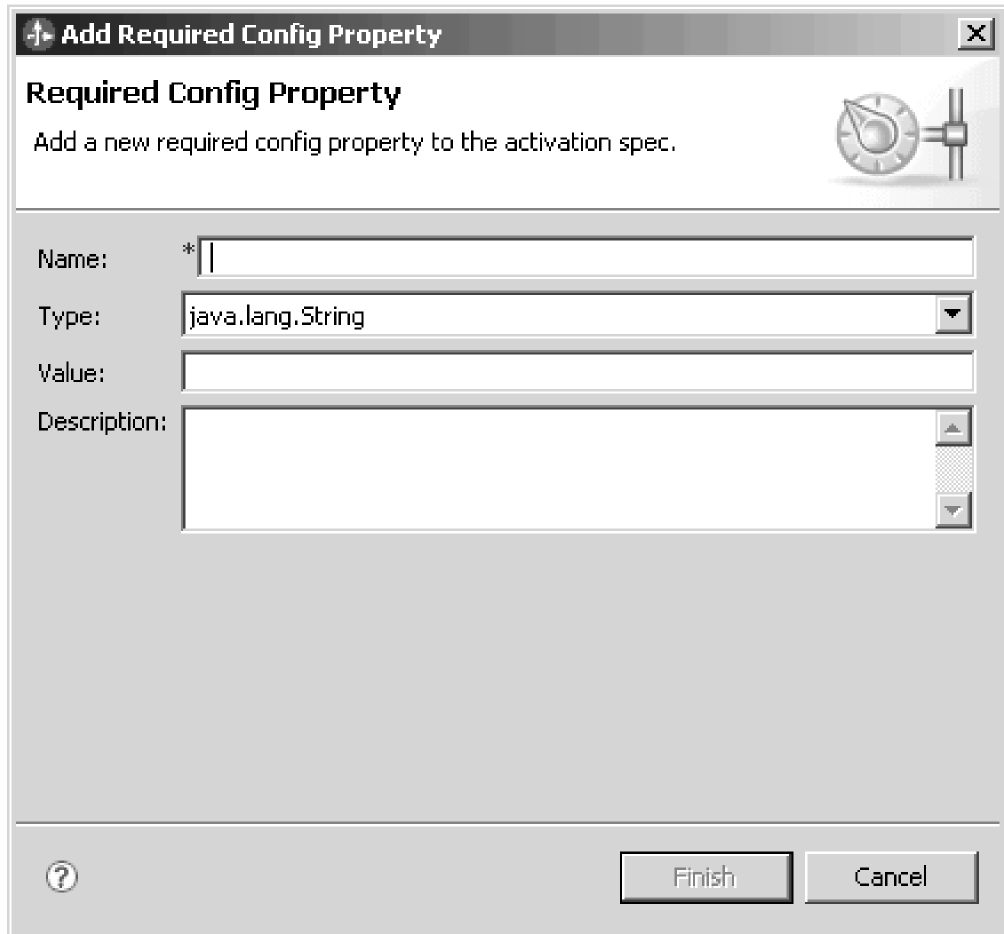
class. Clicking the **Add** button under the list of Message Listeners on the left side of the editor displays the following dialog box.



Add Message Listener dialog

Once a message listener is defined all properties inherited by the specified ActivationSpecWithXid are shown in the properties list directly under the ActivationSpecWithXid Class.

You can specify default values and descriptions for these inherited properties. In addition, you can add, update, or delete your own user-defined configuration properties using the **Add**, **Edit**, or **Remove** buttons on the right side of the editor.



Add Required Config Property dialog

When you add, modify, or delete user-defined properties in this section, the editor creates (or removes) the corresponding Java bean properties in your code. For more information, see Modifying properties.

Modifying deployment descriptor properties

When you modify configuration properties, the Resource Adapter Deployment Descriptor automatically updates source code with corresponding JavaBean properties.

When you define configuration properties, you typically must also add them to the source code as JavaBean properties. The Resource Adapter Deployment Descriptor performs this and other tasks automatically:

- When you add a property using the editor, the editor adds the appropriate field and accessor methods to the Java code for you.
- When you delete a property, the editor removes the field and accessor methods for you.
- When you modify the type of a property, the editor adjusts the field and accessor methods accordingly.

Note: Removing or modifying properties may cause compile errors in your code that you must resolve.

Generated bean properties

The editor maps resource adapter properties to class code. When you modify the resource adapter, the editor performs automatic source code updates.

The table shows the generated code affected when you add, delete, or modify a configuration property:

Class code affected by configuration property change

Configuration property	Affected class code
Config Property (Resource adapter panel)	ResourceAdapter
Connection Definition Property (Outbound adapter panel)	ManagedConnectionFactory ConnectionRequestInfo
Message Listener Property (Inbound adapter panel)	ActivationSpecWithXid

Editing deployment descriptor source

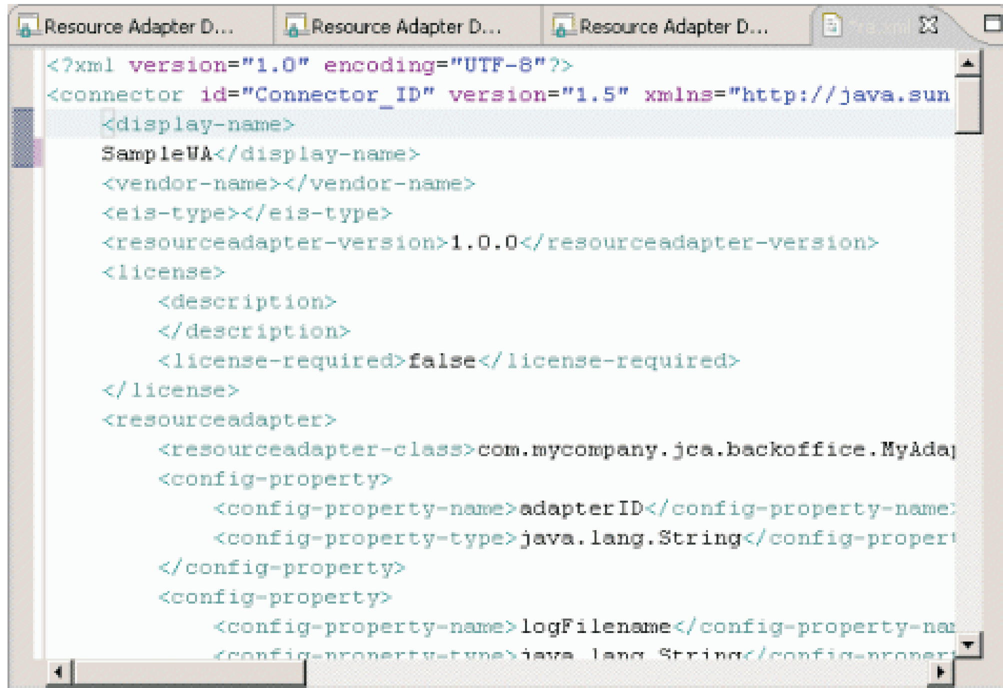
You can view and edit your adapter's deployment descriptor directly.

Although the Resource Adapter Deployment Descriptor editor provides a fully functional and convenient method for viewing and editing the deployment descriptor, you may find it useful to view or edit the ra.xml file directly. The steps to do this are as follows:

1. Make sure that the ra.xml file is not currently displayed in the editor.

   You cannot open the file if it is already open in the editor.
2. From the Project Explorer pane, select the **Deployment Descriptor: Connector** <*your connector name*> file.
3. Right click and choose **Open With** → **XML Source Page Editor**

The XML Source Page Editor displays the ra.xml file.



You can now view or modify the raw ra.xml file using the default editor provided by WebSphere Integration Developer. When you use this editor to save this file, the file is automatically validated against the Resource Adapter XML Schema Definition (or .xsd) file. Any errors are displayed in the Problems pane.

Important: When using the Resource Adapter Deployment Descriptor editor provided by the WebSphere Adapter Toolkit, you can open valid deployment descriptors only. If you attempt to open a deployment descriptor that is not valid (as defined by the schema), the XML Source Page Editor displays the file so that you can fix the problem before attempting to open it in the Resource Adapter Deployment Descriptor editor.

Implementing code from the IBM WebSphere Adapter Toolkit

Foundation Classes implementation overview

To develop a WebSphere resource adapter, you identify the JCA classes for your project, generate subclasses of all the corresponding Foundation Classes, and then modify or override methods as described in the individual class subsections. Subclasses that override methods in the Foundation Classes should invoke the super method so that generic logic is still invoked (unless explicitly indicated otherwise).

The steps shown below provide an overview of how to build an adapter from the Foundation Classes. Note that these steps focus on the major classes to extend. In some cases, you must provide yet another JCA interface implementation to extend a class; when this occurs, locate the appropriate Adapter Foundation class and extend it for your purposes. Typically this requires minor changes only.

1. Decide which configuration properties you need for the resource adapter to support the enterprise information system (EIS):
   a. Identify configuration properties for connecting to the EIS instance for outbound processing (for example, hostname, port).

b. Identify configuration properties suitable for use by a client for a specific outbound connection instance (for example, username, password, language).
c. Identify configuration properties for inbound event processing in general–this will probably be a combination of those you've defined in 1a and 1b for outbound.
d. Establish a list of remaining adapter configuration properties–those not related or relevant to inbound or outbound configurations.

2. Extend class `WBIResourceAdapter` and provide accessor methods for configuration properties defined in 1d. Add these same properties to the resource adapter deployment descriptor.
3. If the resource adapter supports inbound event processing through the event manager Quality of Service (QoS) as described later in this user guide, modify your `WBIResourceAdapter` subclass to implement interface `WBIPollableResourceAdapterWithXid` and provide an implementation of interface EventStore. If you defined additional inbound properties in 1c, beyond what is already defined for class `WBIActivationSpec`, extend `WBIActivationSpecWithXid` and update your deployment descriptor to reflect this new subclass class under the supported activation specs.
4. Extend class `WBIManagedConnectionFactory` and add accessor methods for any properties defined in 1A. Add these same properties to the resource adapter deployment descriptor.
5. Extend classes `WBIConnection` and `WBIInteraction`. In your `WBIInteraction` subclass, provide logic for processing requests (create, retrieve, update, and delete operations). In your `WBIConnection` subclass, simply provide the ability to generate a new `WBIInteraction` instance.
6. Extend `WBIStructuredRecord` and implement the `getNext()` and `extract()` methods. If the adapter is intended to be used in an SCA environment, implement data bindings.
7. Extend class `WBIManagedConnection` and provide logic to physically connect and disconnect to the EIS. In the getConnection methods, simply return the `WBIConnection` subclass created above.
8. If you defined additional connection-specific properties in 1b beyond what is already defined in classes `WBIConnectionRequestInfo` and `WBIConnectionRequest` extend both and add these properties.

See the Adapter Foundation Classes Javadoc for the complete descriptions of the classes.

Data model

In any system where heterogeneous components exchange data, a common data model or object format is crucial. With a common data model, system components know what to send and what to expect in return.

For Websphere Process Server and Websphere Enterprise Service Bus, this data model is called a *business object*. The adapters handle data internally, in a format-independent manner, using the Data Exchange Service Provider Interface (DESPI). In the WebSphere Process Server and WebSphere Enterprise Service Bus environment, a *data binding* produces and consumes the business objects, and communicates with the adapter using the Data Exchange SPI.

The JCA 1.5 specification defines an optional CCI Record model.

The business object data model provides for the following:

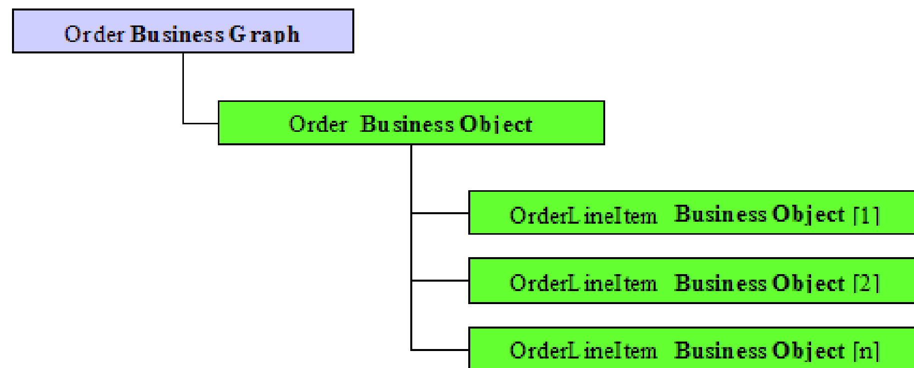
- A full, working implementation, as opposed to the CCI Record model that simply defines interfaces that must be implemented by the adapter developer.
- A built-in support for tracking changes at both the object and property levels, which allows for improved efficiency in processing and reduced bandwidth requirements for exchanging data.
- The business object data model is based upon the open-standard service data object (SDO) model that is supported by IBM and others (visit www.eclipse.org for more information).
- The business object data model aligns well with the larger WebSphere service-oriented architecture (SOA) strategy which, going forward, will better enable interpretability with other WebSphere-based applications.

Relationship of business objects to service data objects

At the technical level, the WebSphere business object model maps directly to the service data object (SDO) model: a WebSphere business graph and business object correspond to an SDO data graph and data object.

A business graph is a top-level structure that defines a single child business object which can itself contain zero or more child business objects.

Using a business graph optional. Use it to provide a *change summary* for applying a delta updates to hold metadata about its child business objects.



Business graph

After-images versus deltas

Two distinct types of business objects, after-image and delta, are used to convey different kinds of information.

After-Image business objects can be thought of as a snapshot of the data in time; they reflect how the EIS entity looks (for inbound events) or is expected to look (for outbound requests). An after-image business object should represent the entire entity structure in the EIS. For example, if an ORDER object with UPDATE verb is sent to the adapter, and the order has two ORDER_LINE children, the object in the EIS should be modified to reflect the input; if the EIS's representation of that object had only one ORDER_LINE, it will have two after the processing has completed.

Delta business objects reflect the specific changes that have occurred (for inbound events) or that the user wants affected (for outbound requests) in the EIS. Each business object contains a changeSummary structure that can store all the pending

changes. For outbound requests, the adapter must interpret the change summary, making all applicable changes to the data. For example, if an ORDER_LINE has been added to an ORDER object, the ORDER_LINE will appear as Created in the change summary. The adapter is responsible for finding that ORDER, and adding the ORDER_LINE to it.

Verbs

A verb is a property of a business graph.

If the data object being processed by the adapter has no business graph, *there will be no verb*.

The *AppplyChanges* function, when passed a business graph, can perform either an afterimage function or a delta update. If the verb is populated, the adapter will perform the operation indicated by the verb. If no verb is present, the change summary will be used to perform a delta update.

When encountering a delta business object, a component can introspect the change summary to determine what actually occurred.

Verbs versus operations

Verbs and operations are similar conceptually but serve different purposes. Operations reflect the functions that an adapter can perform. An operation is directly related to the adapter performing it. By contrast, a verb is directly related to the business object in which it is specified. In general, the verb defined for a business object should match the operation but not always; some operations are unrelated to the verb.

An example helps illustrate the difference between verbs and operations. If you wanted to create a new entity in the EIS using an after-image business object, you would specify an object verb of Create. Accordingly, a Create operation would invoke the adapter. If you invoke an adapter Delete operation with an object that had a Create verb, the adapter would report an error.

However, some operations, specifically those that do not fall under Create, Update, or Delete, do not require verbs. For example, the Retrieve operation of adapters, which has the adapter query the EIS to find an entity that matches the business object, does not expect a verb. This Retrieve operation is unconcerned with what action lead to the business object being created (the information reflected by the verb); rather it is concerned only with retrieving an EIS entity that is reflected by the object.

Note: The only time verbs are part of adapter processing is when the object has a business graph, and the function used is *applyChanges*. In that case, and only that case, the verb dictates the processing to be performed by adapter.

Business object standards

Business object naming:

EIS object names are extracted during the metadata import process. These names often must be modified for use with a resource adapter. Business object names, such as Customer or Address, must reflect the data structures they represent and follow a camel case initial capitalization format.

Convert business object names from EIS-assigned formats to a camel case format (remove separators such as spaces or underscores and capitalize first letter of each word). For example, convert the EIS name, ORDER_LINE_ITEM, to OrderLineItem.

As described in the WebSphere business object specification, name the parent business object graph for the contained business object followed by BG. For example, CustomerBG is the parent object graph for a Customer business object.

Business objects names as well as property names should have no semantic value to the adapter. When you develop your adapter logic, be sure that it is based on metadata as opposed to naming conventions.

Standard operations:

As described by the JCA specification, a resource adapter is generally intended to expose low-level, EIS-specific operations. These EIS-specific operations range from create, retrieve, update, and delete (for database-type applications) to those that are unique to customer EIS instances (for function-based adapters such as SAP).

The WebSphere business object model and a variety of WebSphere components (such as the relationship management service) assume that most adapters support a set of standard create, retrieve, update, and delete operations. This leaves a gap for most adapters where the supported low-level operations do not natively meet the expectations of other WebSphere components. To fill this gap, all adapters should support higher-level create, retrieve, update, and delete operations, depending on what is provided by the EIS. With such support, a Create operation for one adapter follows the same naming conventions and behavior as a Create operation for another adapter. The result is better tooling and consistency in terms of user experience.

The following are the supported, standard top-level operations:

Supported standard top-level operations

Inbound Operation Signatures	Notes

`emitCreateAfterImage<BOType>`
`emitUpdateAfterImage<BOType>`
`emitDeleteAfterImage<BOType>`
	These operations should generate after-image business objects with verbs that match the operation signature; for example, `emitCreateAfterImageCustomer` should generate a Customer object with verb Create.

`emitDelta<BOType>`	This operation should generate a delta business object with a summary depicting the changes that occurred in the EIS.

Outbound Operation Signatures	Notes
applyChanges<BOType> create<BOType> update<BOType> delete<BOType>	These operations can handle delta or after-image business objects. The assumption is that the adapter can consume either type of object or, if not, can convert after-image and delta as required by the business object structure. For applyChanges, the adapter determines the operation (create, update, or delete) based on the top-level verb or the change summary. applyChanges allows users to easily pass any create, update, or delete business objects. The create, applyUpdate and delete operations are specific to one operation. Note that applyChanges, for after-images, should invoke the appropriate <x> operation. **Exceptions**: If the adapter cannot support delta for a given business object type (because it lacks retrieve capability to convert), a signature of applyAfterImage<BOType> may be substituted.
retrieve<BOType>	Retrieve one object based on key values.
retrieveAll<BOType>	Retrieve multiple objects that match some user-defined predicate; this is a query option intended to replace RetrieveByContent. RetrieveAll should always return a top-level container with 0..n matching child business objects. It should never return a single top-level matching business object as with the Retrieve operation.

Standard processing logic:

Each adapter should enable operations that are supported by the EIS.

General guidelines for operations with adapters include the following

- All operations should be atomic: if an operation fails for any reason, the adapter should roll back any partial changes made to the EIS as part of the request.
- For all operations, adapters should never modify the input business object passed by the client (per JCA standards). Instead, if the operation requires the same object passed as input to be returned as output, use WebSphere Business Integration utilities to create, modify, and return a deep copy of the input object.
- If child objects are included in the input business object, the order of child objects should be maintained in the output business object. Doing so enables relationship management service support.
- If an after-image is passed to the adapter as input, an after-image should be returned as output. The same applies for deltas.

- Adapters should follow strict conventions in processing business objects. This includes failing if an entity is marked as updated in the input business object but does not exist in the EIS (rather than attempting to create the entity in the EIS).

isSet property:

WebSphere business objects support an isSet property.

isSet property

The isSet() API of the DESPI InputAccessor determines if a given business object property has been set. When isSet() is false the adapters should ignore the specific attribute while processing the request.

Strict interpretation of requests:

An adapter should always fail if the user provides data that is inconsistent with either the behavior of the adapter or the state of the EIS. This requires more effort from maps or users to ensure that data is appropriate before exposing it to an adapter. That effort also reduces the chance of miscommunication (that is, that the adapter will do something unintended).

If an adapter receives a request in which a business object is marked as created but, in fact, an entity already exists in the EIS with the given key values, the adapter should fail immediately.

Some WebSphere Adapters have historically made a best effort and defaulted to an update in such cases. Such adapters attempted to interpret the request and make every effort to complete it.

ApplyChanges operation:

The applyChanges operation is a *catch-all* operation that enables users to send any create, update, or delete business object to the resource adapter for processing based on the verb.

The applyChanges operation saves effort and simplifies mapping with simple logic: for after-image business objects, the applyChanges operation should look at top-level verb in the business object and then call create, update or delete as appropriate; when applyChanges is invoked without a verb, this is a delta operation. The adapter should read the SDO change summary and perform all the changes indicated in the change summary.

After-image Create operation:

The after-image Create operation generates a new entity in the EIS that matches the data and structure of the input business object. The business object returned by this operation should accurately reflect the newly created entity in the EIS.

Processing overview

The processing of the after-image create operation, which starts at the top-level business object, is as follows:

1. Create an entity in the EIS corresponding to the type of the input business object

2. If the EIS does not generate its own primary key (or keys), insert the key values from the input business object into the appropriate key column (or columns) of the EIS entity.
3. Update the output business object to reflect the values of the newly created EIS entity; this includes any EIS-generated key values or properties marked as having potential side-effects (see property-level metadata).
4. Recursively create the EIS entities corresponding to the first-level child business objects, and continue recursively creating all child business objects at all subsequent levels in the business object hierarchy.

Operation return value

The output written to the output cursor should contain any newly-created key values and other side effects.

Error handling

The `DuplicateRecordException` exception is thrown if the EIS already contains an entity with the same key values as a business object to be created.

The `InvalidRequestException` exception is thrown if any of the following inputs to the operation ate not supported:
- If key values are specified in the input business object but the EIS supports auto-creation only
- If key values are not specified in the input business object but the EIS requires them
- If the top-level verb, if provided, is not Create (assertion optional)

The `EISSystemException` exception is thrown if the EIS reports any unrecoverable errors.

After-image Update operation:

The after-image update operation modifies an EIS entity so that it and its child objects match the data and structure of the input business object. It requires an explicit comparison of the input business object to the EIS system.

Processing overview

After-image update processing is as follows:after-image Update
- Compare the existing business object in the EIS with the input business object and create, update, or delete entities to match the input as follows:
  – If child entities exist in the application, they are modified as needed.
  – Any child business objects contained in the hierarchical business object that do not have corresponding entities in the EIS are added to the EIS.
  – Any child entities that exist in the EIS but are not contained in the business object are deleted from the application.
- Update the output business object to reflect the modified EIS entities; this includes any EIS-generated key values or properties marked as having potential side-effects (see property-level metadata).

Operation return value

Note: When writing the output values to the output cursor, be sure to include any generated keys or other side effects.

Error handling

Error handling behavior includes any and all exceptions thrown by create and delete operations plus the following:

`RecordNotFoundException` is thrown if the EIS does not contain an entity with the same key values as the business object to be updated.

`EISSystemException` is thrown if the EIS reports any unrecoverable errors.

`MissingDataException` is thrown during adapter operations to indicate that not all the necessary information has been provided as required.

`InvalidRequestException` is thrown during adapter operations during adapter operations to indicate poorly formatted data was provided.

After-image Delete operation:

The after-image Delete operation removes an existing entity and any contained child entities from the EIS.

Processing overview

The after-image Delete operation is processed as follows:
1. Perform a recursive retrieve on the input business object to obtain all data in the EIS that is associated with the top-level business object.
2. Perform a recursive delete on the entities represented by the input business object, starting from the lowest-level entities and ascending to the top-level entity; non-contained entities should be left intact although any relationships to deleted objects should be removed if explicitly defined in the EIS.

Note: Adapters should also delete any and all contained children whether or not they are reflected in the input business object. For example, if just a top-level business object is provided with keys and no children, the adapter should still check for contained children in the EIS and delete them.

Operation return value

Since the deletion of an entity in the EIS only requires a return that indicates the success (or failure) of the operation, the goal is to convey this with as little overhead as possible. The Delete operation might or might not return anything. The adapter should handle the case where the output record is initialized with the same metadata as the input cursor. In this case, it is not necessary to populate the output data on `getNext`, apart from the key information.

Error handling

The `RecordNotFoundException` exception is thrown if the EIS does not contain an entity with the same key values as the business object to be deleted.

The `InvalidRequestException` exception is thrown if input to the operation is not supported.

The `EISSystemException` exception is thrown if the EIS reports any unrecoverable errors.

Retrieve:

This operation rebuilds the complete business object hierarchy. The adapter ensures that the returned hierarchical business object matches exactly the database state of the application entity.

The Retrieve operation accepts either an after-image or delta business object. The comparison in either case will be by equality only. Non-key values are allowed as match criteria.

The request business object can contain any of the following:
- A top-level business object but no child objects, even though the business object definition includes children.
- A business object that contains the top-level business object and some of its defined children.
- A complete hierarchical business object containing all child business objects.

The difference between Retrieve and RetrieveAll is that Retrieve is intended to return a single, unique business object that meets user-defined criteria whereas RetrieveAll returns multiple matching business objects. For example, use Retrieve to find Customer where id="abc123" and RetrieveAll to find all Customers where state="NY".

Processing overview

Retrieve processing is as follows:

When the retrieve operation is invoked, it is preferable to retrieve the record information from the EIS and put it into the output structured record, where `getNext()` will populate that information into the output cursor. If it is not possible to retrieve the information from the EIS until `getNext()` is called, it is acceptable to perform the entire retrieve operation inside of the `getNext()` method.

Error handling

`RecordNotFoundException` is thrown if any populated properties in the input business object does not exist in the EIS.

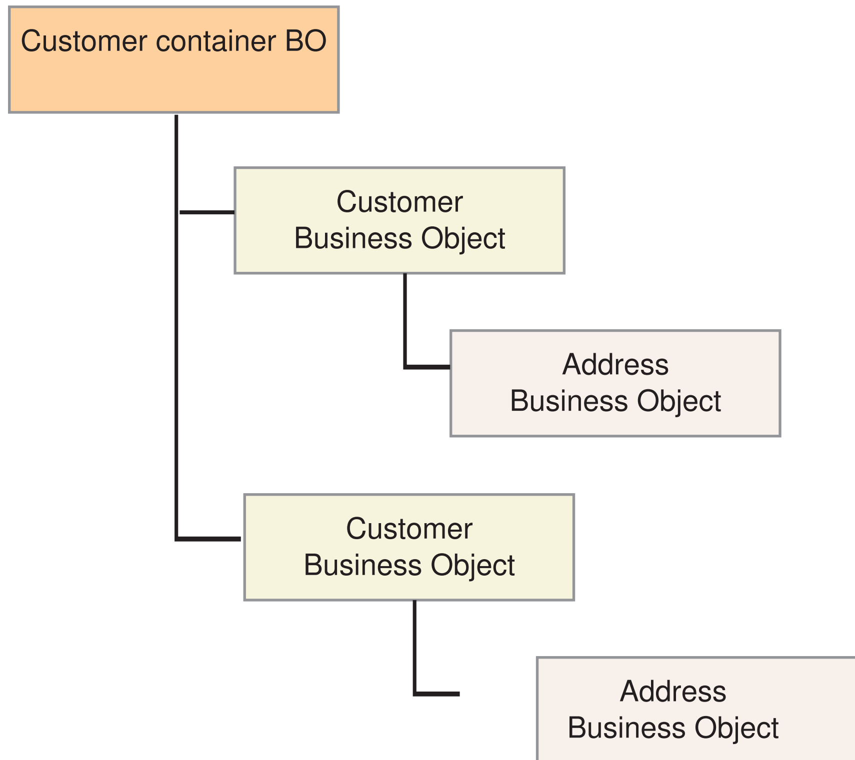
`MultipleMatchingRecordsException` is thrown if more than one record match input criteria.

`EISSystemException` is thrown if the EIS reports any unrecoverable errors.

RetrieveAll:

RetrieveAll returns a batch of records that match the values provided in the request business object. The records are returned as a collection of business objects through a top-level container business object.

CCI clients of resource adapters that support batch results must be capable of recognizing a top-level container and iterating through the child objects that represent the results of the query. The client can then extract any individual business object in the container and deliver it to the rest of the system as with any single business object. This obviously requires the creation of additional business object container structure definitions by the user or during metadata import for each business object type that the user intends to query in batch. The business object container is shown in the illustration.



Business object container

This approach is favored over that of the *JDBC-style* ResultSet support as described in the JCA specification. However, if an EIS provides native ResultSet support or if it makes sense then adapter developers are encouraged to implement the CCI ResultSet interfaces to provide customers with a high-performance alternative to the generic batch retrieval approach described in this document.

The container object is produced by the data binding for use in the Websphere Process Server or Websphere Enterprise Service Bus environment. Internally, the adapter will model the multiple retrieved records as a list of top-level records that can be iterated over using the `getNext()` method of the output Record.

Note: The use of the operation name RetrieveAll rather than RetrieveByContent (as used by the WebSphere Business Integration adapters) distinguishes this operation as a new and clear standard. Support for RetrieveByContent was inconsistent across previous adapters: some would retrieve a single object and return special error codes if there were more objects that matched while other adapters would create special containers to return multiple values.

Note: The `RetrieveAll` operation always returns a result set regardless of how many (if any) matches are found.

Processing overview

RetrieveAll processing is as follows:

RetrieveAll should make the adapter ready to return multiple objects. For each of the objects that will be returned, the ″getNext()″ method will be called. Each call to ″getNext″ should advance the cursor to the next top-level record.

Note: Adapters should check the property MaxRecords in the WBIInteractionSpec instance to determine the maximum number of records to return in order to avoid out-of-memory issues.

Operation return value

The adapter performs a query and retrieves a result set of all objects that match a given set of values. The output object is a container that holds an 0..n objects of the same type as the input object.

Error handling

`RecordNotFoundException` is thrown if any populated properties in the input business object do not exist in the EIS.

`MatchesExceededLimitException` is thrown if the number of hits in the EIS exceeds the value of MaxRecords as defined in the interaction specification. The property MatchCount will contain the actual number of hits that the adapter had in the EIS so that users can either increase their limit or refine their search appropriately.

`EISSystemException` is thrown if the EIS reports any unrecoverable errors.

Custom operations:

Adapters support custom operations that enable more robust means of interrogating or modifying the EIS. Custom operations include Execute (to execute a stored procedure or script) or Lock (to lock an entity in the EIS).

For custom operations, implementations should accept both after-image and delta business objects and simply use the values provided in the data portion of the business graph. For example, if a delta is provided without enough information to perform a custom request operation, the adapter should attempt to retrieve the necessary information from the EIS application when possible or throw an `InvalidRequestException` that expands missing data.

For custom operations, implementations can interpret the data coming in the input cursor in any way that is desirable. Be sure to throw the appropriate exceptions if the input data is insufficient.

Inbound event notification

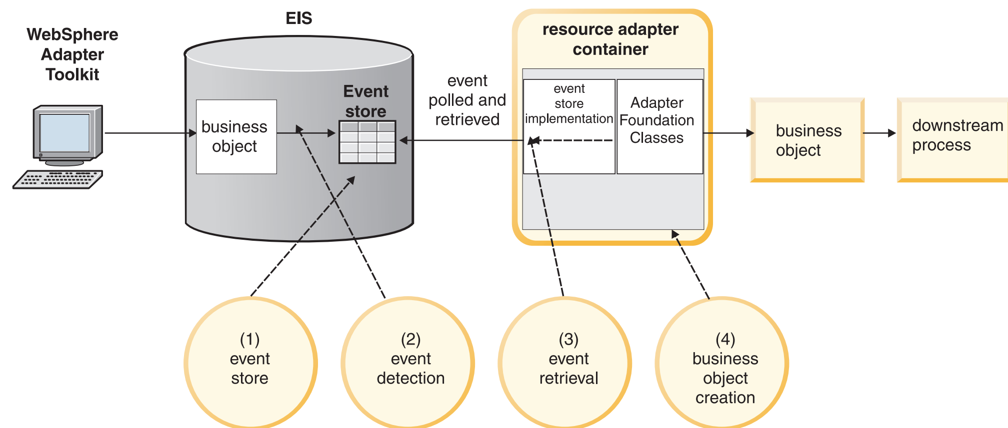
When you enable inbound event-notification, business processes are alerted to changes in, or new information about, an EIS.

Inbound event notification complements outbound request processing, enabling adapters to provide bi-directional communication between business processes and

EIS applications. Depending on the underlying EIS, the business events an adapter generates may span the set of changes that have occurred to a given entity in the EIS, such a customer changing the quantity in their order from 10 to 100, to a complete document or binary payload such as an insurance claim form submitted in XML.

Although each EIS application is unique, most adapters implement inbound event-notification in similar ways:

1. Create an event store in the target EIS to persist changes or other relevant event data that is published by the adapter.
2. Implement an event detection mechanism in the EIS. This mechanism is responsible for detecting any changes of interest (to the adapter) in the EIS and recording them in the event store.
3. Implement an event retrieval mechanism in the adapter that can detect and retrieve events from the event store described in (1) above.
4. Implement a data transformation mechanism in the adapter to convert EIS events to WebSphere business objects for use by target business processes.



Inbound event notification

Using the IBM WebSphere Foundation Classes for inbound event notification

Although not required, use of the IBM WebSphere Foundation Classes is strongly recommended for adapters that need to provide event notification.

Using the Foundation Classes can dramatically simplify the often complicated implementation of event retrieval and publication. The Foundation Classes can automatically track endpoints (the consumers of events) for the adapter, control the polling for and delivery of events, handle recovery of events if the adapter unexpectedly terminates, and assure once-and-only-once event delivery. This allows developers to provide greater quality of service (QoS) in less time and also ensure that behavior across adapters is consistent.

In order to employ the Foundation Classes for event-notification, the adapter and EIS application must meet requirements described in the following sections.

Event Store Requirements

1. Event data must be persistent. Once detected in the event store, an event should remain available there until deleted by the adapter regardless of connection failure or time elapsed.
2. The event store must allow the adapter to both identify and change the state of event records in the event store.

Adapter Requirements

To manage application-specific events, the Foundation Classes require that you provide application-specific logic, if any. To do this, you must:

1. Ensure that any subclass of com.ibm.j2ca.base.WBIResourceAdapter implements interface com.ibm.j2ca.WBIPollableResourceAdapter. This interface allows the Foundation Classes to acquire an EventStore implementation that specifically reflects the EIS application.
2. Provide an implementation of the com.ibm.j2ca.extensions.eventmanagement.EventStore interface. This interface allows the Foundation Classes to manage events in the store without requiring specific knowledge of how and where the event store is implemented.
3. Extend `WBIActivationSpecForXid` to include the base polling properties.

   There is an alternate ActivationSpec, called `WBIActivationSpecForPooling`, that contains two additional properties; Minimum connections, and MaximumConnections. These properties are used by the event manager to establish a connection pool of `EventStoreWithXid` instances. Each instance of `EventStoreWithXid` is treated as a ″connection″.

   If the user sets MaximumConnections to a value greater than 1, and the delivery type is set to a value other than ″ORDERED″, multiple threads will be used in delivery, and each delivery thread can potentially be assigned a discrete connection.

   If you decide to use `WBIActivationSpecForPooling`, keep in mind that ″createEventStore″ can be called multiple times on your adapter, so be sure to handle this appropriately.

Application Requirements

In many cases–and even when your adapter does not implement the Foundation Classes–an application must also be configured or modified before the adapter can use the event-notification mechanism.

Modifications to the application might include setting up a user account in the application, creating an event store and event table in the application database, inserting stored procedures in the database, or setting up an inbox. If the application generates event records, you might need to configure their text. You might also need to configure the adapter to use the event-notification mechanism. For example, a system administrator might need to set adapter-specific configuration properties to the names of the event store and event table.

Assured once-and-once-only event delivery

XA transactions support *once-and only-once* event delivery.

Assured once-and only-once delivery is implemented with an XAResource. The XAResource keeps track of transaction IDs in the event table. The event table contains an XID (string) field. The adapter queries and updates that XID field.

During recovery, WebSphere Application Server calls the resource adapter, queries it for XAResources, and then performs transaction recovery as follows:

- Transactions that the J2EE container rolls back have not been delivered and are marked NEW.
- Transactions that the J2EE container commits have been delivered; these are deleted.

Implementing an event store in the EIS

An event store is a persistent storage area in the EIS application where event records are saved until the adapter can process them. The event store might be a database table, application event queue, email inbox, or any type of persistent store. A persistent event store enables the application to detect and save event records for the adapter even when the adapter is not operational.

Note: Always consider performance implications and scalability when choosing where and how to implement an event store. For example, if you are building an adapter for a database application, an event store represented as a table in the database will most likely perform more efficiently than an event store implemented in an external event inbox.

Event records:

There are no *hard and fast* rules governing the structure or content of an event record in the event store. The goal is to provide enough information for the EventStore interface implementation to successfully generate a business object that represents the event.

Common event record fields

Field	Description
Event Identifier (ID)	A unique identifier for the event
Object Key	The application-specific data that uniquely identifies the entity that occasioned the event.
Business Object Name	The type of the business object that maps to the entity.
Verb	The operation that triggered this event: create, update, delete, and so on.
Timestamp	The time at which the application generated the event.
Status	The status of the event. This is used by the Foundation Classes to track which events are new, in process, or processed.
XID	The transaction ID.

Event Identifier (ID)

Each event requires a unique identifier for tracking purposes. This identifier can be a number generated by the application or a number generated by a scheme that your adapter uses. The event might generate a sequential identifier, such as 00123, to which the adapter adds its name. In such an event ID numbering scheme, the resulting object event ID is *MyAdapterNam*e_00123. Another technique might

generate a timestamp to identify an event, producing an identifier such as *MyAdapterName*_06139833001005.

Object Key

Each event should contain enough key information to enable the adapter event-retrieval mechanism to locate and retrieve the full entity in the EIS for which this event was originally recorded. The structure and content of this key is up to you.

For reference, a key is typically comprised of easily-parsed values such as name-value pairs. For example, Customer_ID=123 or, for a composite key, Model=Widget;Color=CanaryYellow.

Business Object Name

Since the EIS entity identified in each event is converted to a business object instance, the type of the business object must either be specified or derivable at run time by the event-retrieval mechanism.

The most straightforward approach is to include the business object name value in the event record. Specifically, the business object graph type should be included; for example, CustomerBG or OrderBG.

Note: The namespace, which is also required for business object creation, need not be included in the event; the namespace is user-defined and should be retrieved by the adapter from the adapter ActivationSpecWithXid instance.

If, as an adapter developer, you choose to derive the business object graph type, you are strongly advised against mandating any specific EIS entity name to business object type mapping schemes. Business object properties and names are unreliable because they can be modified by users. To avoid such problems, adapters should always process business objects using metadata.

Verb

The verb should reflect the operation that this event represents. For example, if this event reflects the creation of an entity in the application, the verb would be Create. This value should be one of the standard verbs used by adapters (including Create, Retrieve, Update, Delete) and specified in the business object generated for the event. When properly specified in the business object, the verb allows consumers of the business object event to determine what action to take on the event.

Note: The default function selector implementation (WBIFunctionSelector) uses the verb passed in the event business object to determine the appropriate SCA EIS import operation. Be sure that verbs used here correspond to the operations defined by your adapter enterprise metatdata discovery implementation.

Timestamp

The Foundation classes use the timestamp to ensure proper ordering of events. For example, use of a timestamp prevents an event describing the deletion of an Order from being published before an event describing the creation of that same Order. The timestamp should provide detail sufficient to distinguish events occurring close in time.

Status

The event status is used to track the state of an event. It allows the Foundation Classes to distinguish among events that are new from those in process or ineligible.

The adapter must support five different event status values as described in the table below. All events generated by the event detection mechanism in the EIS should be in the initial state of New. Only the Foundation Classes, through the `EventStore.updateEventStatus` method, change event status.

Possible Event Statuses

Event Status	Description	Foundation Class Constant
New	The event is ready to be processed.	`NEWEVENT`
In Progress	The adapter is processing this event. Note that an event in this state may or may not yet be delivered.	`INPROGRESS`
Processed	The adapter successfully processed and delivered the event.	`PROCESSED`
Failed	The adapter was unable to process this event due to one or more problems.	`FAILED`
Unsubscribed	The adapter processed the event but found no interested subscribers	`UNSUBSCRIBED`

XID

The XAResource uses this string field to track transaction IDs in the event table. The adapter queries and updates that XID field. During recovery, WebSphere Application Server calls the resource adapter, queries it for XAResources, and then performs transaction recovery based on the XID.

Event object:

An event object is an instance of the com.ibm.j2ca.extensions.eventmanagement.Event class as defined in the Foundation Classes. An event object is the common representation of an event in the Adapter Foundation Classes and it is populated from an event record in the getSpecificEvent(String eventId) method described in the Implementing the EventStore Interface section.

Event object fields

Field	Description
eventID	Corresponds to the Event Identifier field of the event record
eventKeys	Corresponds to the Object Key field of the event record

Field	Description
eventType	Corresponds to the Business Object Name field of the event record
timeStamp	Corresponds to the Timestamp field of the event record
eventStatus	Corresponds to the Status field of the event record

Event detection:

Events detection mechanisms reflect the sources that trigger them: user actions in the application, batch processes that add or modify application data, or database administrator actions.

When an event detection mechanism is set up in an application and an application event associated with a business object occurs, the application must detect the event and write it to the event store.

Event detection mechanisms are application dependent. Some applications provide an event detection mechanism for use by clients such as adapters. The event detection mechanism may include an event store and a prescribed way of inserting information about application changes into the event store. For example, one type of implementation uses an event message box that receives messages from the application when it processes an event of interest to the adapter. The adapter application-specific component periodically polls the message box for new event messages.

Other applications have no built-in event detection mechanism but have other ways of providing information when application entities change. If an application does not provide an event detection mechanism, you must use whatever mechanism is available to extract information on entity changes for the adapter. Among those mechanisms are database triggers, exit calls to programs that write to event stores, or extracting information from flat files that aggregate application changes.

In all cases, the event detection mechanism should ensure data integrity between an application event and the event record written to the event store. For example, the generation of an event record should not occur until all required data transactions for the event have completed successfully.

Steps Involved

In general, an application event detection mechanism should take the following steps:

1. Detect an event on an application entity.
2. Create an event record.

To create the record, the event detection mechanism should:

1. Generate a unique event identifier (ID).
2. Set the object key to the primary key of the application entity.
3. Set the verb to the action that occurred in the database.
4. Set the event timestamp.

5. Set the name of the WebSphere business object complexType that corresponds to this application entity
6. Set the event status to New.

Implementing event retrieval in the adapter

The careful work of implementing event retrieval in the adapter uses two Foundation Classes interfaces. The goals are setting up event polling and a safe, reliable connection to the event store.

Adapters that employ the Foundation Classes for event retrieval must meet the following requirements:

1. Implementing interface com.ibm.j2ca.base.WBIPollableResourceAdapterWithXid in any WBIResourceAdapter subclass.
2. Implementing interface com.ibm.j2ca.extensions.eventmanagement.EventStoreWithXid

The first requirement identifies (for the Foundation Classes) the adapter for event polling. If this interface is implemented, the Foundation Classes automatically begin checking for and publishing events as dictated by polling-related configuration properties such as PollPeriod and PollQuantity and as specified by active adapter endpoints.

The second requirement, implementation of the EventStoreWithXid interface, typically requires the most (adapter) development effort. As the location and structure of each event store is application-specific, the EventStoreWithXid interface provides the Foundation Classes with a common means of querying and modifying an event store.

Implementing an EventStore Interface

An EventStore implementation is responsible for establishing and managing a connection, if necessary, to the underlying EIS application. The EventStore implementation should be thread-safe because it will be accessed on multiple threads in Unordered delivery mode.

The table below describes the methods that each EventStore implementation must provide:

EventStore methods

Method	Description
`public void setEventTransactionID(Event event, XidImpl xid) throws ResourceException, CommException`	This method should store the xid in the Event table in the same row specified by event. xid.toString() will serialize the Xid for easy storage.
`public Xid[] getPendingTransactions() throws ResourceException, CommException`	This method should return the XIDs for events that have an associated XID, but are still in NEWEVENT status.
`public Event getEventForXid(XidImpl xid) throws ResourceException, CommException`	This method should return the event associated with the given Xid.

Method	Description
`ArrayList getEvents(int quantity, int eventStatus, String[] typeFilter)`	This method enables the adapter to determine if there are any new events available or old events that need re-sending. Implement this method to query the event store and return a list of event instances (up to the limit specified by the**quantity** parameter) that have a status matching the value of parameter **eventStatus**. The order of events returned in **ArrayList** should reflect the sequence of events as intended for publication. If this EventStore implementation supports filtering as specified by method **implementsFiltering**, this method should inspect the value of parameter **typeFilter**. If **typeFilter** is not null, the method should return events that match the type(s) specified only. If **typeFilter** is null (or filtering is not supported), the method should simply return events of all types.
`boolean implementsFiltering()`	This method provides the Foundation Classes with information on the capability of the EventStore implementation. If it can filter events by type in method **getEvents**, the implementation should return `true`; otherwise it should return **false**.
`Event getSpecificEvent(String eventId)`	This method should reconstruct a complete event object for the event identifier. This will most likely require that the EventStore implementation query to retrieve the missing information from the event record in the EIS (for example, object type, status, and so on).
`Object getObjectForEvent(Event event)`	The EventStore implementation should inspect the event passed and return a business object instance reflecting the changed entity in the EIS application. For example, if the event specifies an object type of CustomerBG and a key value of CustomerID=123, this method might be expected to return a CustomerBG instance populated with the values from customer 123 in the EIS.
`void deleteEvent(Event event)`	The EventStore implementation should delete the event record identified from the underlying EIS event store.
`void updateEventStatus(Event event, int newstatus)`	The EventStore implementation should modify the event record identified with the provided status code.

Transaction Support Methods

If the implementation of the event store supports transactions, the EventStore implementation should provide access to that transaction control using the following methods:

EventStore transaction control methods

Method	Description
`boolean isTransactional()`	Is the event store transactional? If so, this method should return `true`.
`void commitWork()`	This method should commit the pending transaction. It is required only if transactions are supported.
`void rollbackWork()`	This method should rollback any uncommitted work. It is required only if transactions are supported.

Possible event store implementations

An event store is usually implemented in a database, however, any structured persistence mechanism could be used

Implementing the event store with a database:

If an EIS application incorporates a database, you can use the database to store event information.

Where to store events

To locate the event store in the EIS application database, you must create a new, separate WebSphere event table there. The table then functions as the event store for event records. Each column of the table would reflect one of the fields mentioned in Table 1; each row would reflect a unique event.

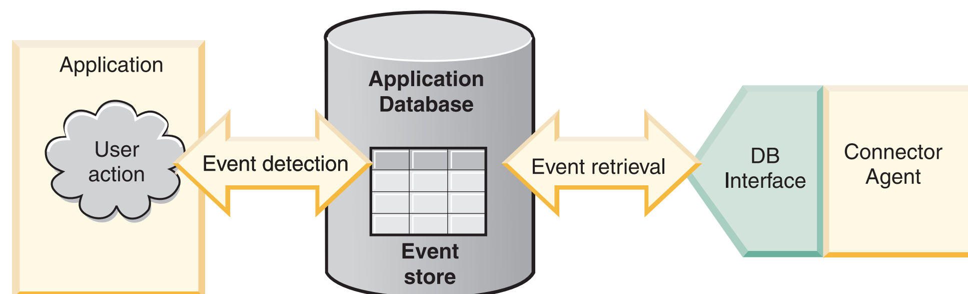
Implementing event detection

If the application has no built-in method for detecting events and the database that the application is running on provides database triggers, you could implement row-level triggers to detect changes to application tables. When changes occur in one of these tables, the triggers would write new event records to the event table.

Note: If possible, avoid full table scans of existing application tables as a way of determining whether application tables have changed.

Retrieving events

The EventStore implementation would need to employ the available database APIs to gain access to the contents of the event table.



Event store implementation

Function selector:

Function selectors map resource adapter events to corresponding SCA export function names.

The WebSphere Adapter component that exposes resource adapters as SCA components requires what is known as a function selector. This selector maps events generated by resource adapters to a SCA export function name. For example, an adapter may generate an after-image Customer event with top-level verb Update, which the user expects to be published using the function emitCreateAfterImageCustomer.

```
public interface FunctionSelector {
   public String generateFunctionName(Object[] argObjects) throws MetadataException;
}
```

The `StructuredDataFunctionSelector` class looks at metadata within the StructuredRecord to generate a function name. It will create a function name as follows ″emit[OperationName]AfterImage[RecordName]″, where OperationName is an operation stored in the record as the operationName property, and RecordName is the value stored in the recordName property.

For more information on the FunctionSelector interface, see the Metadata Discovery Specification.

For example, if the `StructuredDataFunctionSelector` received an event such as CustomerBG with TopLevelVerb Create and containing a business object of type Customer, the WBIFunctionSelector class would generate a function name such as emitCreateAfterImageCustomer. For the same business graph with no TopLevelVerb, the function name emitted might be emitDeltaCustomer.

Error handling for events:

Event error handling depends on the delivery type and the kind of endpoint involved.

If the endpoint throws an exception during delivery, the event manager will stop delivering events to that endpoint, and the timer task for polling stops. If the delivery type is ORDERED, the remaining events polled in that cycle are not delivered until the event with the error is processed. If the delivery type is UNORDERED, the event manager attempts to deliver the remaining events in the current poll cycle. When the endpoint is taken offline and reactivated, the event(s) in which the error occurred is re-delivered, and normal delivery of subsequent events resumes. If the endpoint is transactional and the transaction rolls back, the event manager responds as if the endpoint threw an exception.

If the implementation throws an exception during the getObjectForEvent call (for retrieval of the full event), the event manager marks the status of that event in the event table as`ERROR_PROCESSING_EVENT`. If the delivery type is ORDERED, the polling task stops until the endpoint is reactivated. If the delivery type is UNORDERED, the polling task continues.

Inbound callback event notification

An EIS application's capability to call the adapter directly, by registering a listener, is known as a *callback*. If your application supports the callback capability, you can make use of *callback event notification* support in the adapter foundation classes.

When you enable inbound callback event notification, business processes are alerted to changes in, or new information about, an EIS. The phrase *callback* refers to the ability of the EIS system to directly notify the adapter or business processes of a change, as opposed to the polling mechanism used in event notification.

Callback event notification complements outbound request processing, enabling adapters to provide bidirectional communication between business processes and EIS applications.

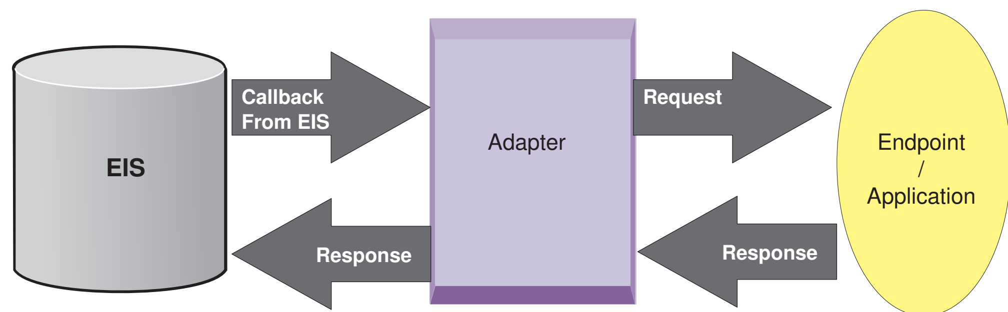
Generally, in a callback scenario, the adapter will need to setup event listeners to receive callback events from the EIS. Callback event processing could be either synchronous (REQUEST-RESPONSE) or asynchronous (ONE-WAY).

Request and response callback events

A request and response callback event is a synchronous operation in which the EIS sends a callback call to the adapter and waits for the adapter to respond to the call.

Since the EIS expects a response from the adapter, event delivery to multiple endpoints cannot be supported.

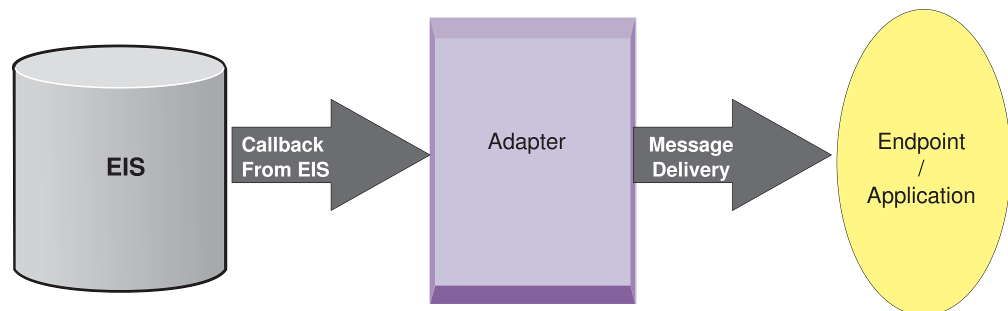
The following is an illustration of synchronous callback event processing.



One way callback events

One way callback events are *asynchronous* operations in which the EIS sends an event to the adapter and then goes on with its processing, not waiting for the adapter to send a response back.

The following is an illustration of asynchronous callback event processing.



Using the IBM WebSphere adapter foundation classes for inbound callback event processing

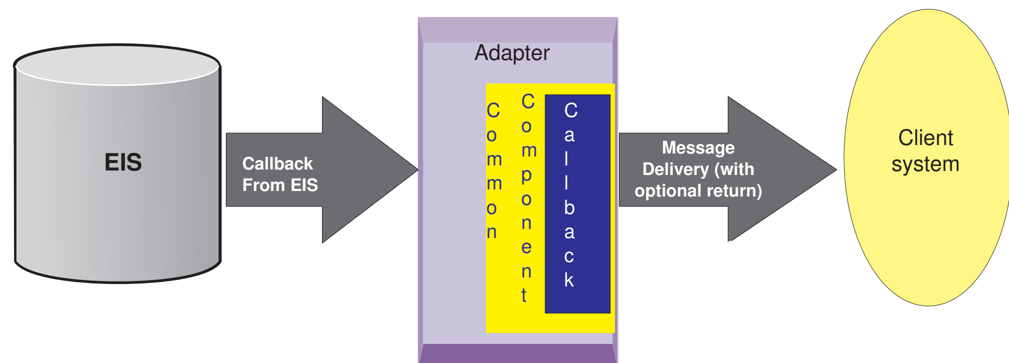
The adapter foundation classes can automatically track endpoints (the consumers of events) for the adapter, control the event pick up and delivery of events, handle recovery of events if the adapter unexpectedly terminates, and assure once-and-only-once event delivery.

This allows developers to provide greater quality of service (QoS) in less time and also ensure that behavior across adapters is consistent.

Although not required, it is a recommended practice that you use IBM WebSphere adapter foundation classes (AFC) for adapters that have to provide callback event notification.

Adapter foundation classes make the process of callback event delivery in an assured manner easier for the developer creating the adapter, by providing the event delivery API called the `CallBackEventSender`.

The following diagram depicts the usage of adapter foundation classes (common component). The `CallbackEventSender` API is integral to the adapter foundation classes.



Callback event sender

CallbackEventSender in com.ibm.j2ca.extension.eventmanagement.external package provides four public methods.

CallbackEventSender in `com.ibm.j2ca.extension.eventmanagement.external` package provides following four public methods.

- public void `sendEventWithNoReturn(Record, InteractionSpec)` throws `WBISendFailedException`
- public Record `sendEventWithReturn(Record, InteractionSpec)` throws `WBISendFailedException`
- public void `sendEventWithNoReturn(GenericEvent, Record, InteractionSpec)` throws `WBISendFailedException`
- public Record `sendEventWithReturn(GenericEvent, Record, InteractionSpec)` throws `WBISendFailedException`

These methods end up invoking different methods in the message-driven bean (MDB). The MDB will implement several interfaces, including InboundListener, and MessageListener.

The `sendWithReturn` methods invoke *onMessage* on the InboundListener. This method delivers the Record it received from the listener thread. Here the difference is the onMessage method will be invoked on the InboundListener to deliver the Record to the endpoint. The method would return "Record" as returned by onMessage method.

The `sendWithNoReturn` method invokes *onNotification* on the InboundListener. The sendEventWithReturn() method is not supported incase of multiple endpoint factories configured, hence will throw an appropriate exception. Also, if there are failures during the sending event when multiple endpoint factories configured, a consolidated exception stack trace will be thrown with details like which endpointfactory failed and for what reason.

Callback event sender constructors:

CallbackEventSender provides four different constructors to facilitate different types of client invocation.

The following information describes the usage of the four constructors:

- CallbackEventSender(ArrayList, EventPersistence, XAResource, ActivationSpecWithXid, LogUtils)

  The complete constructor which takes an array of endpoint factories and supports XA transaction with event persistence updates. If the input ActivationSpecWithXid is valid and not null, event would be delivered by calling target method on the inbound listener with ActivationSpecWithXid as additional argument.
- CallbackEventSender(ArrayList, ActivationSpecWithXid, LogUtils)

  Constructor used for event delivery without XA transaction and event persistence support.
- CallbackEventSender(MessageEndpointFactory, EventPersistence, XAResource, ActivationSpecWithXid, LogUtils)

  This constructor does take the same arguments as (1) except that it takes one MessageEndpointFactory instead of an array.
- CallbackEventSender(MessageEndpointFactory, ActivationSpecWithXid, LogUtils)

  A slight variation for constructor (2) with just one MessageEndpointFactory argument instead of an array.

Callback event processing for basic delivery

When the event is created at the EIS end, configured adapterListener gets notified and it in turn instantiates CallbackEventSender. Here adapterListener decides which method to invoke out of the four defined.

To implement callback mechanism, adapter must have an EndPointManager class. The class needs to maintain the relationship between the activationSpec and MessageEndpointFactory arguments. Here is a snippet of EndPointManager class showing how the callback mechanism is implemented.

```
public class EndpointManager {

 /**
 * inner class which maintains pairs of mef and activationspec
 */
      public static class EndpointPair {
       public MessageEndpointFactory mef;

       public ActivationSpec activationSpec;
```

```
       public EndpointPair(MessageEndpointFactory mef, ActivationSpec activationSpec) {
            this.mef = mef;
            this.activationSpec = activationSpec;
        }

         public boolean equals(Object o) {
            if (!(o instanceof EndpointPair)) {
                return false;
              }

             EndpointPair other = (EndpointPair) o;
return other.mef.equals(this.mef) && other.activationSpec.equals(this.activationSpec);
                  }
       }

// adds new endpointPair to the list
public void addEndpoint(MessageEndpointFactory mef, ActivationSpec activationSpec)
throws ResourceException {...}
}
```

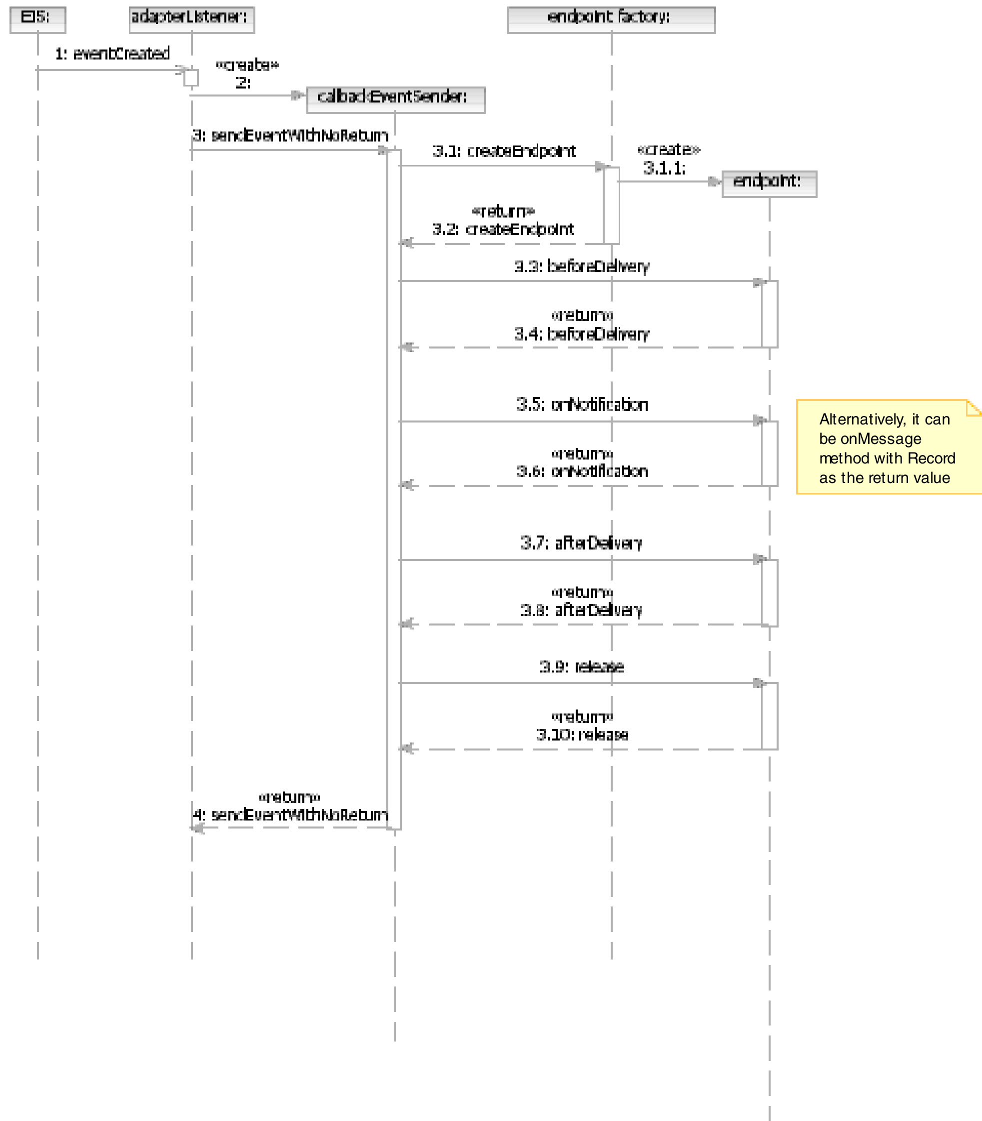
The adapter listener gets all MessageEndpointFactories for the current activationSpec and registers that with the CallbackEventSender instance.

```
EndpointManager epManager =  ((ResourceAdapter)aSpec.getResourceAdapter()).getEndpointManager();
EndpointPair[] endpoints = epManager.getEndpoints(this.aSpec);

if (!isSynchronous && aSpec.getAssuredOnceDelivery().booleanValue())
{// XA Delivery
 callbackEventSender = new CallbackEventSender(endPointList,
   eventRecMngr.getEventPersistance(), XAres, aSpec,
   logger.getLogUtils());
 } else {// Non XA delivery
   callbackEventSender = new CallbackEventSender(endPointList,
   aSpec, logger.getLogUtils());
}

 /* Refer to the Callback event sender constructors section for more
information.
Note: The adapter listener creates the worker threads to take care of calling
and getting responses from the callBackEventSender method. */
```

Once the program control gets into CallbackEventSender, it checks how many EndpointFactories are configured for the current instance of adapter. If there is more than one, then it delivers the event by creating endpoints for each of them and invoking either onNotification or onMessage method on the endpoint with out any XA transaction. Finally, it would call the release() method on the end point to free the endpoint hence the application server can add it to endpoint pool. Finally it would call release() method on the end point to free the endpoint hence the application server can add it to endpoint pool.

Also it invokes beforeDelivery() and afterDelivery() methods on the endpoint as defined by the JCA functional specification.

XA transaction will come into picture only when the adapter is configured with ONE EndpointFactory. The following sequence diagram depicts the callback event processing for basic delivery.



Callback event processing for event delivery with XA transaction

To provide data integrity and to make sure events are not delivered more than once, which would cause errors in the downstream system in the integration scenario, the invention provides a mechanism to achieve once-and-only delivery and the same is accomplished using XA transaction.

When assured delivery is required, the basic flow described in *Callback event processing for basic delivery* becomes more complex. For more complex scenarios, CallbackEventSender creates an instance of XA to bring the delivery under a new transaction. The beforeDevliery() call to endpoint is the starting point of XA transaction and it will last till afterDelivery(). If problems occur within this transaction scope, a proper rollback mechanism will ensure data integrity is maintained.

XA transaction will be active and supported only when the adapter is configured with one EndpointFactory.

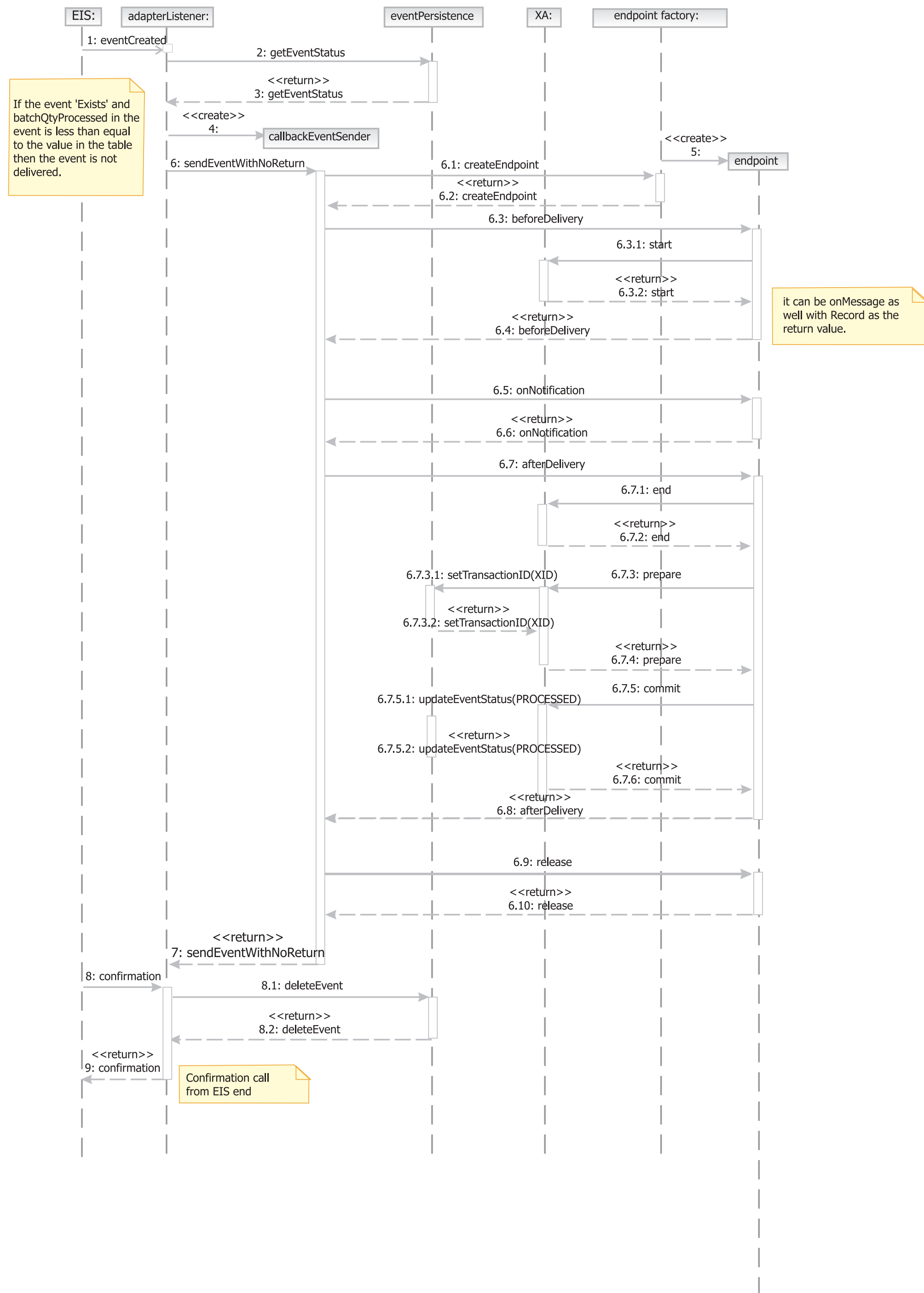
When the adapter signals that it has completed delivery, the transaction manager will then call ″end″, ″prepare″, and ″commit″ to complete the requirements outlined in the XA transaction protocol. When the ″prepare″ call is made, the XA implementation will call ″setTransactionID″ on the eventPersistance implementation; the eventPersistance implementation will store the transaction XID in the event table. When the ″commit″ call is made, the XA implementation will call ″updateEventStatus″ on the eventPersistance implementation to set the status in the event table to ″COMMITED″. This is done for every event that was retrieved. After all events have been delivered and successfully marked ″COMMITTED″.

When the event is created at the EIS end, configured adapterListener gets notified and it in turn instantiates CallbackEventSender. Here adapterListener decides which method to invoke out of the four defined. Lets consider the adapter calls sendEventWithNoReturn() as shown in the sequence diagram.

Once the program control gets into CallbackEventSender, it checks how many EndpointFactories are configured for the current instance of adapter. If there is more than one, then it delivers the event by creating endpoints for each of them and invoking either onNotification or onMessage method on the endpoint with out any XA transaction. Finally it would call release() method on the end point to free the endpoint hence the application server can add it to endpoint pool.

Also it invokes beforeDelivery() and afterDelivery() methods on the endpoint as defined by the JCA functional specification.

XA transaction will come into picture only when the adapter is configured with ONE EndpointFactory. The following sequence diagram depicts the callback event processing for event delivery with XA transaction.



Callback event processing for event recovery

When there is a failure in the event processing as part of system recovery, the adapter is able to recover the unprocessed events by implementing the once-one-only delivery mechanism.

During real time event processing if any component of the business integration system fails then the adapter must process the events that are not completed, and not process the events that are completed. This ensures that once-one-only delivery mechanism is implemented when the system recovers from a failure.

When the container starts, it calls the getXAResources() method on the adapter to get all the associated XA resources. The adapter then instantiates the appropriate XA resource and returns it back to the container.

The JCA container now calls the recover() method on the returned XAResourceImpl to get all the pending transactions from the configured event persistence using the getPendingTransanctions() method. Depending on the transaction state, the container calls either the rollback() or the commit() method on the XAResourceImpl to update the status of the event to NEWEVENT or PROCESSED on event persistence.

After connecting to the EIS, the adapter starts the adapterListener. The EIS then triggers the adapterListener for any new event(s) and the adapterListener in turn calls the CallbackEventSender with the same flow explained in the basic delivery and delivery with XA transaction sequence diagrams. The following sequence diagram depicts the callback event processing for event recovery.

<<return>>
121: eventCreated

Outbound support

Outbound support enables application components to execute operations on an EIS and retrieve the results. Input data passed by the application component is processed by the adapter to make changes or call functions on the underlying EIS.

Issuing outbound requests to a WebSphere adapter is no different than interacting with any other JCA adapter as described in the JCA specification. The basic idea is as follows:

1. A CCI client (for example, an EJB or other business process) looks up a connection factory for the adapter using a JNDI service provided by the application server.
2. The CCI client requests a resource adapter connection from that factory.
3. The CCI client then uses that connection to pass data to, and receive data from, the underlying EIS.

The parts of this process that vary among adapters involve the data structures exchanged and the operation-specific parameters passed.

The data structure for all WebSphere resource adapters for outbound requests is a WebSphere business object wrapped in a `WBIRecord` implementation.

The parameters of the operation for any JCA adapter are defined through an adapter-specific `InteractionSpec` instance; this class can contain `0..n` properties that specify details about the operation to perform. For WebSphere resource adapters, a default WBIInteractionSpec class has one property: FunctionName. Invoking components set the operation to perform in the FunctionName property. (This is different from the verb that is defined in the actual business object). You are strongly encouraged to use this InteractionSpec class. For example:

```
WBIConnection conn;
WBIRecord input;
WBIRecord output;
...
Interaction ix=conn.createInteraction();
WBIInteractionSpec ixSpec=new WBIInteractionSpec();
ixSpec.setFunctionName(WBIInteractionSpec.CREATE);
output = ix.execute(ixSpec, input);
```

Application sign-on

The Adapter Foundation Classes can use either container-managed or component-managed authentication or sign-on.

The process of connecting to a back-end application, such an EIS, usually requires some type of authentication. In a JCA environment, application authentication is known as *sign-on*. It can be performed in one of two ways:

- When using container-managed sign-on, the JCA container is responsible for providing sign-on credentials. Sign-on credentials are passed from the JCA container to the resource adapter as an instance of `javax.security.auth.Subject`.
- When using component-Managed sign-on, the adapter client performs a programmatic sign-on by passing explicit security information, such as *username* and *password*, to the resource adapter using the CCI ConnectionSpec implementation.

The *res-auth* element in the application component deployment descriptor specifies the sign-on method. The only valid values for this element are `Container` or `Application`.

Certain back-end systems support reauthentication. Reauthentication is the process of changing the security context of an existing physical connection. If reauthentication is supported by the back-end application, you can set the *reauthentication-support* element of the resource adapter deployment descriptor to `true`. Otherwise it must be set to `false`.

Although it does not define a specific authentication mechanism, the JCA architecture supports two commonly used mechanisms: BasePassword authentication and Kerberos authentication. Use the *authentication-mechanism-type* element of the resource adapter deployment descriptor to specify which type is supported.

To support authentication, resource adapters extend `WBIManagedConnection` as follows:

1. Implement method `WBIConnection(PasswordCredential pc, boolean reauthenticate)`.
2. Extract and use the credentials provided in the `PasswordCredential` instance that is passed; the Foundation Classes provide values from either the subject for container-managed sign-on or a `WBIConnectionSpec` instance for component-managed sign-on as appropriate.
3. (If you don't support reauthentication, skip this step.) Check if the reauthentication flag is true and reset the connection authentication appropriately; this flag should be set to true only if the developer updates the deployment descriptor.
4. Return a `WBIConnection` instance.
5. (Optionally) override `isConnectionInfoOverwriteable()`. This value is used to determine whether the `WBIConnectionRequestInfo` already associated with the `ManagedConnection` can be overwritten by another parameter that satisfies the match condition. By default, this method returns `false`. If you can support changing some connection parameters without destroying the connection (for example, language), override and return `true`.
6. If you override `isConnectionInfoOverwritable`, consider overriding the boolean `matchConnectionRequestInfo (WBIConnectionRequestInfo)`

7. The ConnectionManager may call `getConnection(Subject, ConnectionRequestInfo)` on a `ManagedConnection` where the passed `ConnectionRequestInfo` does not match the `ConnectionRequestInfo` already associated with the `ManagedConnection`. The default implementation of this method performs a property-for-property comparison between the two `ConnectionRequestInfo` instances. It returns `true` if an exact match is found, otherwise `false`. If the match criteria is something different, the resource adapter developer may override this method with a suitable implementation.
8. Return `true` if the `WBIConnectionRequestInfo` is deemed a match with the `WBIConnectionRequestInfo` already associated with this `ManagedConnection`; otherwise return `false`.

Implementing outbound support

You enable outbound support by providing an EIS-specific implementation of a resource adapter. This requires extending the Adapter Foundation Class implementations of common client interfaces (Connection, Interaction, and Metadata) and the ManagedConnection and ManagedConnectionFactory interfaces.

WBIManagedConnectionFactory:

A javax.resource.spi.ManagedConnectionFactory instance manages the creation and configuration of physical connections to the underlying EIS. Specifically, `WBIManagedConnectionFactory` implements the`ManagedConnectionFactory` and `javax.resource.spi.ResourceAdapterAssiociation` interfaces.

Configuration Properties

EIS-specific subclasses should specify boxed JavaBean-compliant accessor pairs (e.g., `setValue(Integer i)` rather than `setValue(int i)`). The accessor pairs get and set EIS-specific outbound configuration properties and logic. This is the means by which property change events reach property change listeners with corresponding updates to the ResourceAdapter (RA) deployment descriptor, thereby making the JCA container aware of available properties. Properties defined in this class are generally intended for use by the WBIManagedConnection implementation when connecting to the EIS.

Note: Support for the ResourceAdapterAssociation interface, which cannot be used in unmanaged environments, is optional for the JCA container. Accordingly, when defining properties, assume that ManagedConnectionFactory will not have access to the ResourceAdapter bean or any properties defined at the ResourceAdapter level. While the ManagedConnectionFactory can check for and use properties at the ResourceAdapter level as defaults when available, any properties defined at the ResourceAdapter level and used by the ManagedConnectionFactory should be optional, contain default values embedded in the ManagedConnectionFactory, or exposed in the ManagedConnectionFactory so that users can specify values if the ResourceAdapter bean is not available.

Subclass methods to implement

1. `Object createConnectionFactory(ConnectionManager)`

   This method is called by the JCA container to enable the CCI clients to generate handles to the physical EIS connection. EIS-specific subclasses should implement this method to return an EIS-specific factory instance which is a subclass of WBIManagedConnectionFactory.

```
public Object createConnectionFactory(ConnectionManager connMgr) throws
  ResourceException
{
 return new TwineBallConnectionFactory(connMgr, this);
}
```

2. `ManagedConnection createManagedConnection(Subject, ConnectionRequestInfo)`

   This method is used by the JCA container to acquire a physical connection to the EIS instance. Subclass implementation should return a EIS-specific ManagedConnection instance which is a subclass of WBIManagedConnection.

   ```
   public javax.resource.spi.ManagedConnection createManagedConnection(
      javax.security.auth.Subject subject,
      javax.resource.spi.ConnectionRequestInfo connReqInfo)
      throws javax.resource.ResourceException
   {
        return new TwineBallManagedConnection(this, subject,
                     (WBIConnectionRequestInfo) connectionRequestInfo,
          this.getResourceAdapter());
   }
   ```

WBIManagedConnection:

WBIManagedconnection is an abstract class which implements javax.resource.spi.ManagedConnection. A javax.resource.spi.ManagedConnection instance represents a physical connection to the underlying EIS. The WBIManagedConnection instance implements methods that enable the JCA container to monitor its status and manage its life cycle.

Subclass methods to implement

1. `public ManagedConnection(WBIManagedConnectionFactory mcf, Subject subject, WBIConnectionRequestInfo)`

   This constructor should connect to the EIS instance and maintain a handle to the connection as a property in the ManagedConnection. This method also must call ″super″ to ensure that the credentials are present for later matching.
2. `ManagedConnectionMetaData getMetaData()`

   Create and return a new instance of WBIManagedConnectionMetadata by passing the EIS specific details such as product name, product version, maximum connections and user name.

   ```
   public ManagedConnectionMetaData getMetaData()
                 throws ResourceException
   {
     return new WBIManagedConnectionMetaData("TwineBall","0.2",200,userID);
   }
   ```
3. `Object getWBIConnection(PasswordCredential, boolean)`

   If reauthentication is supported, implementation should perform an EIS-specific sign-on based on the credentials passed. Otherwise, it should return a CCI handle to this managed connection.

```
public java.lang.Object getWBIConnection( javax.resource.spi.security.PasswordCredential arg0,
                              boolean reauthenticate)
     throws javax.resource.ResourceException
{
    return new TwineBallConnection(this);
}
```

4. `destroy()`

   This method should close this connection to the EIS and release any resources.

Best practices

- Each ManagedConnection instance should encapsulate at most one connection to the EIS.
- Since there may be more than one Connection instance for each ManagedConnection instance, resource adapter developers should implement private contracts between their WBIManagedConnection subclass and their WBIConnection/WBIInteractionsubclasses to ensure that access to the underlying EIS connection or API is performed in a thread-safe manner.

  If the EIS API does not support concurrent access by multiple connection handles, concurrent access should not be denied. Instead, the implementation should ensure that one and only one handle can access the EIS at a time (through synchronization blocks, wait/notify patterns, and so on).
- At the start of any EIS-specific method implementation, developers should always invoke `super.checkValidity()`; this method checks the state of the ManagedConnection instance to ensure that it has not been closed, encountered an error, and so on.
- If possible, always employ eager class initialization of the physical connection to the EIS; do not assume that getConnection is the first method invoked. The JCA container may need to access the XAResource, LocalTransaction or other features of the managed connection for recovery, and so on, before any client even issues a first request.

WBIConnectionFactory:

`WBIConnectionFactory` implements the ConnectionFactory interface. Subclasses should implement the constructor only. As javax.resource.cci.ConnectionFactory, this interface enables clients to request connections to an EIS.

Subclass methods to implement

```
public TwineBallConnectionFactory(ConnectionManager connMgr,
                                  WBIManagedConnectionFactory mcf)
{
   super(connMgr, mcf);
}
```

WBIConnection:

WBIConnection implements the `Connection` interface. An instance of this interface, such as javax.resource.cci.Connection, represents a client connection handle to the underlying EIS connection. A client obtains this connection by calling the getConnection method of the ConnectionFactory instance.

Subclass methods to implement

1. Implement the constructor that takes a ManagedConnection and call the Super class constructor that associates this connection handle with the ManagedConnection.

   ```
   public TwineBallConnection(WBIManagedConnection managedConnection)
                              throws ResourceException
   {
     super(managedConnection);
   }
   ```
2. Create an EIS-specific interaction instance that enables clients to invoke functions on the underlying EIS.

```
public Interaction createInteraction() throws ResourceException
{
  return new TwineBallInteraction(this);
}
```

Note: If you want to provide your own implementation of ConnectionMetadata, you must override method WBIConnection#getMetadata.

javax.resource.cci.ConnectionSpec:

Clients use a javax.resource.cci.ConnectionSpec instance to pass request-specific connection properties to the getConnection method of the ConnectionFactory.

To add EIS request-specific properties, the resource adapter should implement the ConnectionSpec interface directly. The sample below extends WBIConnectionRequestInfo to inherit the properties userName and password, and then adds its own EIS-specific properties.

```
public class TwineBallConnectionSpec extends WBIConnectionRequestInfo implements
  ConnectionSpec {
 private boolean xa;
 public TwineBallConnectionSpec(String userid, String password, boolean xa)
 {
         setUserid(userid);
         setPassword(password);
         this.xa = xa;
     }
}
```

WBIInteraction:

A javax.resource.cci.Interaction instance enables client components to execute EIS-specific operations. WBIInteraction implements an interaction interface to provide implementations for noncritical methods. Subclasses implement the execution interfaces.

Subclass methods to implement

Record execute(InteractionSpec ispec, Record inputRecord)

execute(InteractionSpec ispec, Record inRecord, Record outRecord)

The inRecord is the input record, and outRecord is the output record. The difference between the two is that the output record is passed in, so the code in interaction.execute updates that output record instead of creating a new record to return.

Executes an EIS operation represented by the InteractionSpec and returns an output Record. The following example makes use of the command patterns to simplify processing.

```
public class TwineBallInteraction extends WBIInteraction {
    private CommandManager commandManager;
    private Interpreter interpreter;
    private TwineBallConnection connection;
    private TwineBallCommandFactory factory;

    public TwineBallInteraction(WBIConnection connection) throws ResourceException {
     super(connection);
         this.connection = (TwineBallConnection) connection;
     interpreter = new Interpreter(this.getLogUtils());
         TwineBallResourceAdapter resourceAdapter =
```

```
    this.connection.getResourceAdapter();
         ObjectNaming objectNaming = new ObjectNaming(resourceAdapter);
         factory = new TwineBallCommandFactory(objectNaming);
 commandManager = new CommandManager(factory,
              this.connection.getEISConnection(), this.getLogUtils());
    }

public Record execute(InteractionSpec ispec, Record inRecord) throws
         ResourceException {
        WBIRecord wbiRecord = (WBIRecord) inRecord;
 String functionName = ((WBIInteractionSpec) ispec).getFunctionName();
 WBIInteractionSpec interactionSpec = (WBIInteractionSpec)ispec;
 factory.setMaxRecords(interactionSpec.getMaxRecords());
 Command topLevelCommand =
  commandManager.produceCommands((WBIRecord) inRecord,
                                 functionName);

        topLevelCommand.getClass().getName());
 DataObject returnDataObject = null;
        try {
             returnDataObject = interpreter.execute(topLevelCommand);
        } catch (ResourceException e) {
             log(e);
             throw e;
     }
     WBIRecord outRecord = new WBIRecord();
     if (functionName == WBIInteractionSpec.RETRIEVE_ALL_OP) {
  outRecord.setDataObject(returnDataObject);
     } else {
             outRecord.setDataObject(returnDataObject.getContainer());
        }
        return outRecord;
    }
}
```

WBIInteractionSpec:

A javax.resource.cci.InteractionSpec instance contains properties that identify the operation to perform on the EIS.

`WBIInterationSpec` implements InteractionSpec and provides functionName and maxRecords properties. Client components set these properties to provide information to the resource adapter about the EIS operation to perform. For more information, see the Javadocs for the Adapter Foundation Classes.

EIS specific resource adapter implementations need not extend this class unless they have more EIS operation-specific properties to be added to the InteractionSpec.

WBIConnectionRequestInfo:

A javax.resource.spi.ConnectionRequestInfo instance enables a resource adapter to pass request-specific EIS data structure on a connection request (ConnectionManager.allocateConnection). Client components can set these using connection request properties. `WBIConnectionRequestInfo` implements ConnectionRequestInfo. See the Adapter Foundation Classes Javadocs for more information on this implementation.

Configuration properties

WBIConnectionRequestInfo provides *userName* and *password* properties. Using connection properties, subclasses can add their own EIS-specific properties. These

properties should not change the configuration of the EIS.

javax.resource.cci.ConnectionMetadata:

A javax.resource.cci.ConnectionMetadata instance provides information to the client components about the underlying EIS of a resource adapter. Client components can use the javax.resource.cci.Connection.getMetadata() interface to retrieve connection-specific EIS metadata.

The WBIConnection class implements a subclass ManagedConnectionMetadataWrapper whose constructor takes a ManagedConnectionMetadata instance and implements ConnectionMetadata. WBIConnection also implements the getMetadata() interface to retrieve the ManagedConnectionMetadata from the ManagedConnection instance and then constructs ConnectionMetadata. Hence EIS-specific resource adapter implementations can use this foundation class implementation without having to implement their own ConnectionMetadata instance.

Implementing transaction support

A transaction is an isolated interaction with the EIS. Transaction support allows users to ensure that multiple operations on the EIS are performed as atomic units and are not impacted by other simultaneously occurring operations from other EIS clients.

Note: Transactions can be supported in an adapter only if the underlying EIS supports transactions.

EIS application transactions typically rely on one of two commit protocols: the one-phase or two-phase commit protocols. The one-phase commit protocol allows a client to demarcate the beginning and end of transactional operations with a single EIS application. The two-phase commit protocol, which is a superset of the one-phase protocol, enables transactions to span multiple, heterogeneous EIS systems. Accordingly, applications that support the one-phase commit protocol are often said to support local transactions while those that support the two-phase commit protocol are said to support global, or XA, transactions.

While an adapter can expose support for either or both protocols, the underlying EIS must ultimately provide the support. By this token, you would not attempt to implement XA support in your adapter if your underlying EIS application inherently lacked transaction support.

Once you have determined that your EIS supports transactions, you must make several modifications to your adapter to implement the support.

1. Update Your Adapter Deployment Descriptor Property TransactionSupport, as described in the JCA 1.5 specification, supports three values: `NoTransaction`, `LocalTransaction`, and `XATransaction`. You must specify the appropriate value for the level of support you intend to provide. If your adapter supports XA, specify `XATransaction` support but also implement the local transaction features described below. (The JCA specification prescribes that any adapter supporting XA should also support local transactions.)
2. Update your adapter-specific construction of WBIResourceAdapterMetadata to reflect support for local transactions. ResourceAdapterMetadata#supportsLocalTransactionDemarcation should return `true`.

3. Override method WBIManagedConnection.getLocalTransaction() and, if XA support is provided, method WBIManagedConnection.getXAResource().

Wrap either or both of the LocalTransaction or XAResource instances returned by these methods with a WBILocalTransactionWrapper or WBIXATransacxtionWrapper instance, respectively. These wrappers provide extended diagnostics for troubleshooting and also help adapters determine whether or not to autocommit requests. According to the JCA 1.5 specification, a resource adapter must autocommit transactions when being used outside the context of a transaction. To help the managed connection determine if it is involved in a transaction, these wrappers act as thin delegation layers, monitoring the sequence of calls to determine whether a transaction is active. At the beginning of a transaction, the wrappers call method setEnlistedInATransaction(true) on the WBIManagedConnection instance; upon commit or rollback, the wrappers set this same property to `false`. By then checking the status of the transaction via method isEnlistedInTransaction on the super class, a WBIResourceAdapter subclass can quickly determine whether it should be automatically committing transactions or not when modifying the EIS.

Note: When overriding methods, do not invoke the super implementations of these methods since the Adapter Foundation Classes simply throw exceptions for these methods.

Example of an XA-enabled adapter implementation of WBIManagedConnection

```
public class FooManagedConnection extends WBIManagedConnection
{
// just get the XAResource from your EIS and return the wrapper
                              public XAResource getXAResource() {
XAResource eisXAResource = this.eisXAConnection.getXAResource();
XAResource wrapper = new WBIXATransactionWrapper(eisXAResource,this);
return wrapper;
  }

  // here's an example of a potentially transacted call on the EIS. Point
  // is that adapter should always check whether it's enlisted in a
  // container-managed transaction or whether it should handle transaction
  // on its own
  private void updateRecord(int id,int value) {
 if(!this.isEnlistedInTransaction())
  this.eisConnection.beginTransaction();

 eisConnection.updateRecord(id,value);

 if(!this.isEnlistedInTransaction())
  this.eisConnection.commitTransaction();
 }
}
```

Using command patterns

Command patterns simplify adapter development by providing generic logic for dealing with hierarchical data structures.

Command patterns:

To enhance uniformity across adapters for outbound processing, support for command patterns is provided by the CommandManager API in the Adapter Foundation Classes.

Adapters are responsible for creating, updating, retrieving, and deleting (CRUD) records in the EIS system based on the structure described by the incoming metadata and the content in the incoming cursor. The command pattern approach is recommended for handling the generic processing of CRUD operations for After-Image as well as Delta scenarios.

For example, if an incoming cursor represents an after-image update, the adapter must take steps to update the EIS such that the corresponding object in the EIS matches the structure and contents of the cursor request. To accomplish this, the adapter retrieves the structure in the EIS system, then compares it to the incoming cursor. The adapter then performs the operations necessary to make the EIS system match the input. These operations are typically performed as the comparisons occur. This makes adapter processing potentially quite complex. A command pattern capability abstracts this functionality into generic logic, thereby saving adapter developers time and effort.

The command pattern breaks down a hierarchical update into a hierarchy of small sub-commands. These sub-commands are passed to an interpreter, which retrieves and executes the code necessary to perform the sub-command on that particular EIS system. This makes it possible for the adapter developer to deal with operations on single-tier entities without having to walk the structure and compare. This has the potential to simplify adapter construction greatly because the comparison routines can be generic.

Advantages of the command pattern include:
- Code Reuse: The only code that the adapter developer would need to write would be the EIS-specific operations: the comparator and interpreter code would be common components.
- Performance consistency: Because adapters make use of common components, developers can rigorously define the after-image update process, making a variety of adapters work more consistently.
- Adapter ″phantom″ mode capability: If operations must be performed, the interpreter can easily be turned off, with the contents of the command hierarchy dumped to a file instead.

Command Manager:

The Command Manager is a utility designed to reduce complexity when dealing with delta and snapshot hierarchical objects. You use the Command Manager to consolidate the code necessary to deal with these objects, breaking down the structure into nodes, then creating commands that can deal with each node.

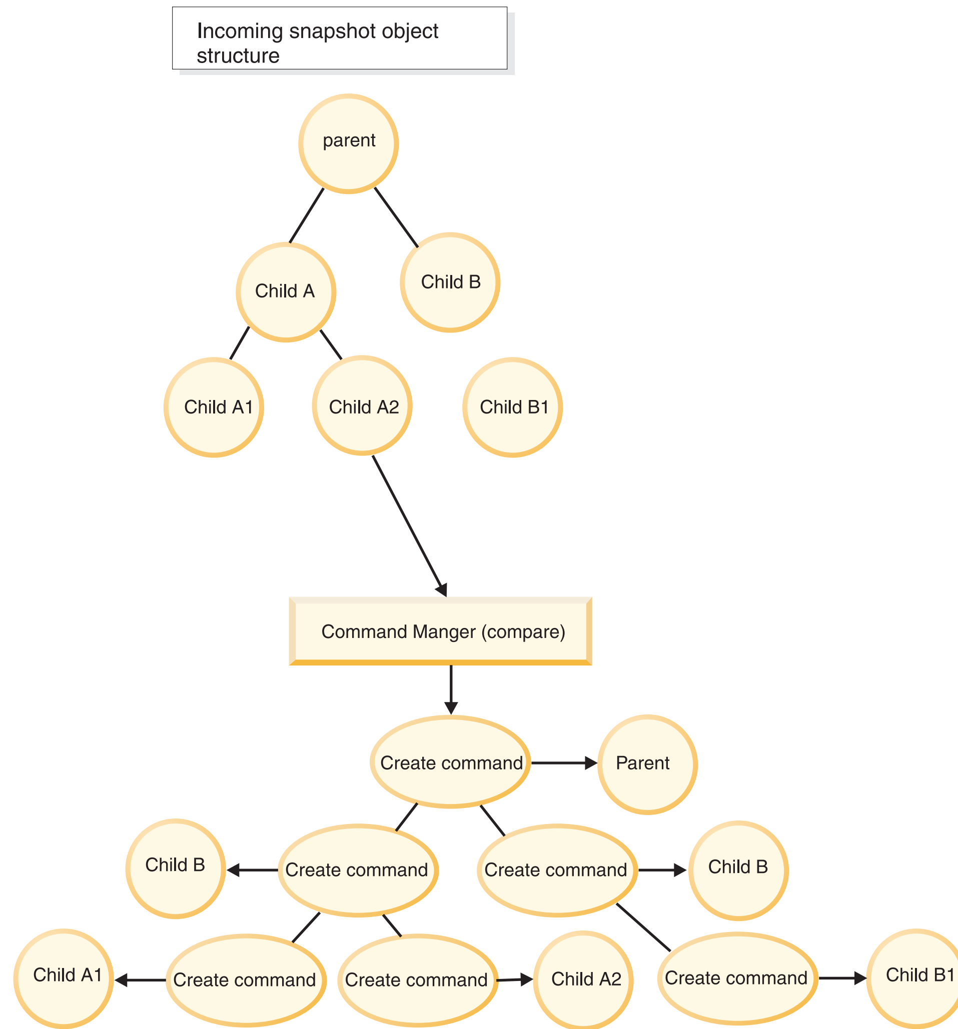
Snapshot objects

Consider the simple case of a snapshot Create operation involving a structure that you want to create inside the EIS. The Command Manager creates a command structure composed of Create commands that is based on the incoming structure.

This command structure will then be executed by the interpreter. As the interpreter executes each individual command, the assigned objects are created.

With a snapshot Update, the Command Manager must retrieve the object from the EIS, compare it to the incoming structure and create a command structure that can change the data in the EIS to match the incoming structure.

Consider the following scenario: Child B1 is in the EIS, but is not in the incoming structure. Child B1, then, must be deleted. The Command Manager will execute a Retrieve command to build the structure as it appears in the EIS, then compare this structure to the incoming object tree. The comparison finds that child B1 is deleted. Accordingly, the resulting command structure has a Delete command in that position.



Command Manager Retrieve scenario

Note: This behavior is dependant on the ability of the Command Manager to determine child keys. Be sure to mark the child key fields in the business object definition using the appropriate Application Specific Information.

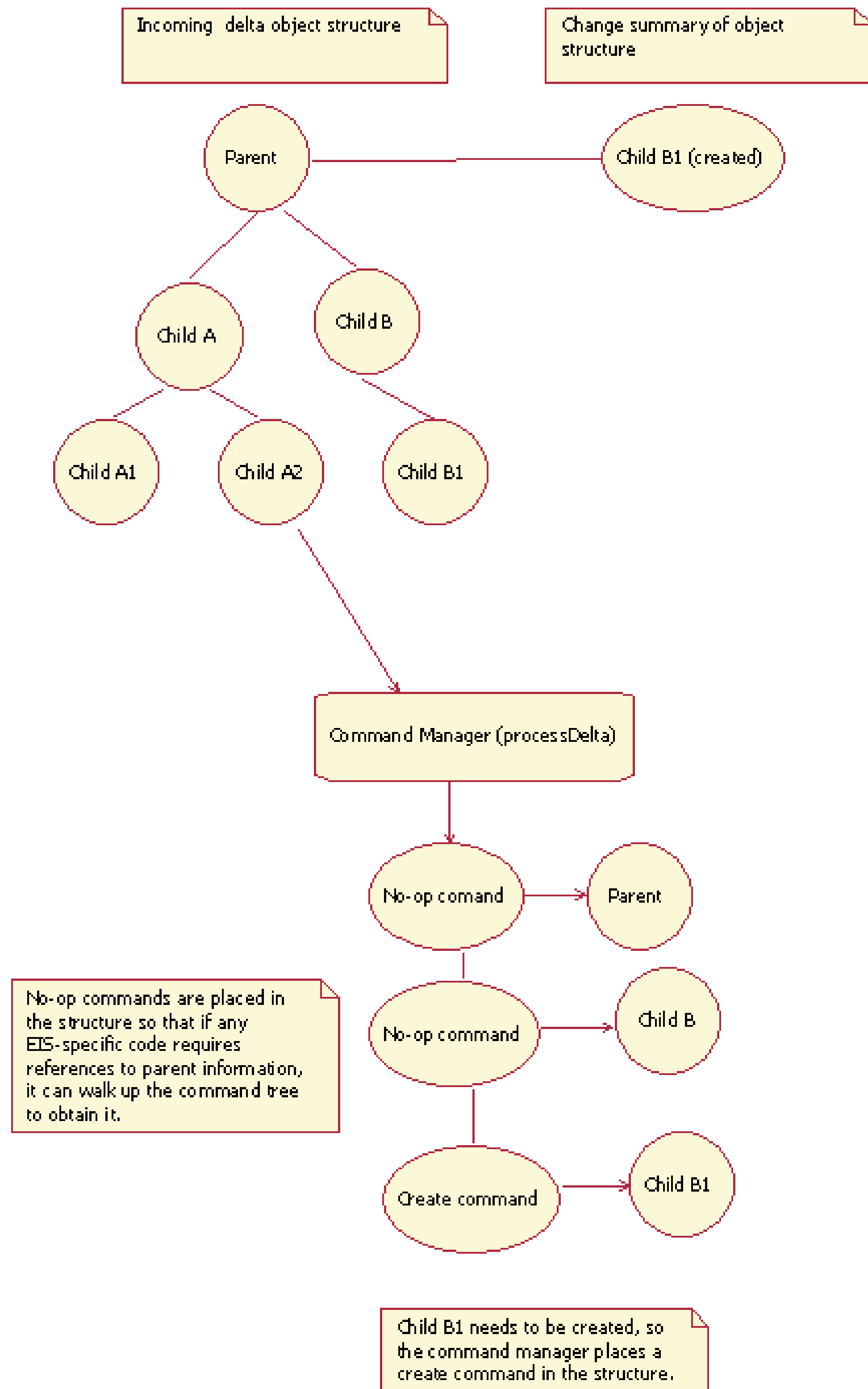
Command Manager Delete

Delta objects

Delta processing is only relevant for service data objects (SDO).

The Command Manager functions in a similar way when processing delta objects. Instead of retrieving and comparing, the command manager reads the change summary for the intended changes. Note that since a child object might be changed without any change to the parent object, and since many EIS systems require a parent pointer in order to process children, the command manager generates NO_OPERATION commands for the untouched parents of changed child objects.

Suppose, as in the example below, that child B1 is created and is part of the change summary. The resulting command structure will contain a Create command for child B1, and will have NO-OPERATION parents linking it back to the top level parent.



Command Manager Create

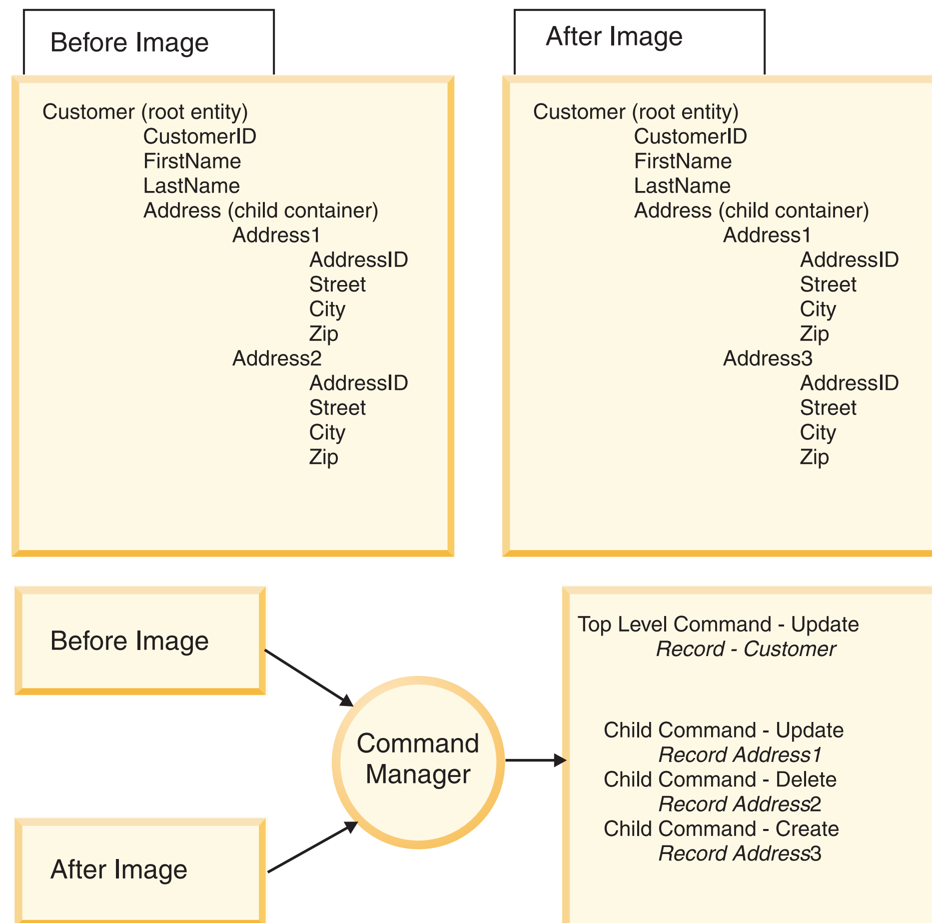
When it processes this structure, the interpreter will execute the No-op commands as well as the Create command. In general, the no-op commands should not modify data in the EIS system.

After-image processing:

The Command Manager, based on the CommandManager API, implements the command pattern capability. This utility simplifies the work of comparing before- and after-image data.

The Command Manager is based on the CommandManager API of the Adapter Foundation Classes. For after-image processing, the Command Manager compares the cursor record structure inside the EIS (the before-image) with that of the incoming after-image cursor. Based on the comparison, the Command Manager constructs a hierarchical representation of commands that must be executed to make the before-image object match that of the after-image. The command hierarchy is passed to an interpreter, which executes each command in the order of execution required for the declared operation. The order of execution for Create and Update is top-to-bottom; for Delete the order is bottom-to-top.

As shown in the following figure, the Command Manager relieves you of the task of developing and testing comparison routines.



Command Manager simplifies before and after comparisons

As shown in the upper portion of the figure, the input to the Command Manager is a before image and an after image. The Command Manager creates a top-level command representing the operation for the top-level incoming cursor. As processing continues, sub-commands are added at the child object level and so on to the top-level incoming cursor, as shown in the lower portion of the figure.

The Command Manager provides a first class support for Create, Retrieve, Update and Delete operations using static variables defined in the `WBIInteractionSpec` class:

WBIInteractionSpec.CREATE
WBIInteractionSpec.UPDATE
WBIInteractionSpec.DELETE
WBIInteractionSpec.RETRIEVE

You can use the Command Manager for other supported operations, too. It creates a command pattern hierarchy with the same operation at all the levels in the in the cursor hierarchy. Specifically, you would configure the CommandFactory for Object type operations with `isOOType()` returning `true`. If `isOOType()` returns `false`, the Command Manager would create only one command in the command hierarchy for the top-level cursor.

For other non-CRUD operations, it may not make sense to use the command pattern capability. For example, if the operation supported is XXX, and if XXX simply executes a function call at the top-level cursor only, there is no need to apply the XXX operation to all child cursors.

As it begins to process children of the root object, the Command Manager iterates through the properties for the top-level cursor. For each property which is of type containment, the Command Manager attempts to construct a hashset of the key values for each child. Every entity, root or child, must have a primary key, which is how the Command Manager distinguishes records.

The Command Manager constructs two sets, A and B. Set A will be the set of primary keys in the before-image child container. Set B is the set of primary keys in the after-image container.

- The set A-B contains extra children. These will be Deleted.
- The set B-A contains missing children. These will be Created.
- The set A intersect B contains children to be updated. These will be updated.

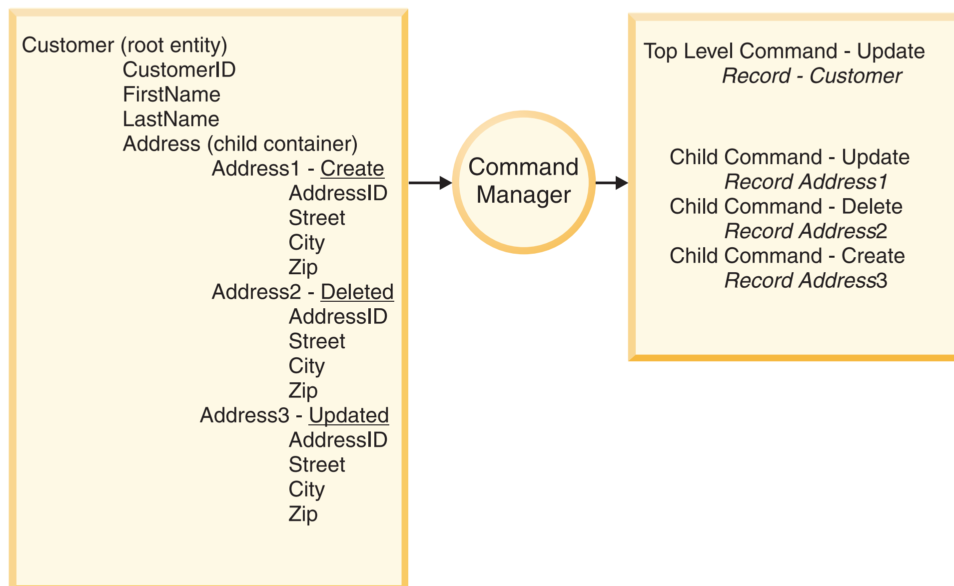
For each record in each set for "extra" and "missing" children, the Command Manager will recursively follow the child record and all its children, for each entity adding to the top level command the child command representing the child cursor and its corresponding operation.

For children to be updated, the before- and after-image data are passed back to the Command Manager. This allows the Command Manager to recursively process any children that those children might contain. When the Command Manager reaches the bottom level of both structures, the command structure is complete, and the Command Manager returns the top level command.

Delta processing:

The Command Manager processes delta structures in a manner analogous to that of after-image data. The difference is that, for delta objects, comparative data is extracted from service data object change summaries.

When the input service data object (SDO) represents an Update operation with a delta structure, the Command Manager will look up the BOChangeSummary to find the respective operations on each child SDO. For example, the following figure shows a delta SDO input with underlined text representing the operation derived from the change summary.



Command Manager extracts change summaries

Command Manager interpreter logic:

An order attribute establishes the sequence by which the interpreter executes commands.

The interpreter is given a top level command to execute. Each command has an order associated with it that is set when the command is created. The order attribute of each command may be `BEFORE_PARENT` or `AFTER_PARENT`. If the order attribute of a child command is `BEFORE_PARENT`, that command will execute before the parent; if the order attribute of a child is `AFTER_PARENT`, that command will execute after the parent.

- If an object is marked as created and then as deleted in the ChangeSummary, the adapter will not take any action.
- If an object is marked as created and then updated in the ChangeSummary, the adapter will perform a Create.
- If an object is marked as updated and then deleted in the ChangeSummary, the adapter will perform an Update.

Implementing Command Manager:

You implement commands for each type of operation supported by the adapter, a command factory to support instances of each operation, and calls to the Command Manager.

You will need to implement the following:

1. Command implementations for each command type ("Retrieve", "RetrieveAll", "Create", "Update", "Delete", and "NoOperation").
2. A command factory implementation that will create instances of these EIS-specific commands.
3. An implementation of Interaction.execute() that calls the Command Manager.

Command implementations:

You use an `execute()` method to implement commands. Extending an abstract base command with operation-specific commands supports the coding benefits of command patterns.

`execute()` is the method that implements each command. This method should perform the appropriate operation for the associated node. If a Create command is associated with child B1, for example, `execute()` is the method that calls the EIS API to create that child.

```
public void execute(InputCursor obj, Type metadata) throws ResourceException
{EisRepresentation eis = EisAPI.insert(toEisFormat(obj, metadata));
}
```

Note: Be careful with deleted children. The SDO `getContainer()` does not return the original parent in a delta operation. Use the `getParentDataObject()` method, available in the command object, to alleviate this problem.

The Retrieve and RetrieveAll commands must create the entire object structure and return it because the input object is unaware of child objects present in the EIS.

It is a common pattern to have an abstract base command for your EIS, and then have the operation-specific commands extend that. This way, if all your commands need similar data, you can reduce your coding effort.

Note: Note: The WebSphere Adapter Toolkit will not generate the `execute()` methods for individual commands. You will want to override the base command execute method to put in the logic you want.

Command factory implementations:

The command factory creates operation-specific command instances and establishes when the instance is executed.

Given an operation name and metadata, the command factory is responsible for creating command instances. The command factory must also set the execution order that specifies when a command should be executed in relation to its parent. This value can be either `BEFORE_PARENT`, or `AFTER_PARENT`.

The interface you must implement is as follows:

```
public CommandForCursor createCommand(String nodeLevelOperation) throws ResourceException;
```

The code to implement a command factory will resemble the following (where TwineBall is the name of the EIS).

```
public CommandForCursor createCommand(String functionName, Type metadata) throws ResourceException {

TwineBallBaseCommand command = null;

try {
```

```
if (functionName.equals(NodeLevelOperations.CREATE_NODE)) {
command = new TwineBallCreateCommand();
} else if (functionName.equals(NodeLevelOperations.DELETE_NODE)) {
command = new TwineBallDeleteCommand();
} else if (functionName.equals(NodeLevelOperations.UPDATE_NODE)) {
command = new TwineBallUpdateCommand();
} else if (functionName.equals(NodeLevelOperations.RETRIEVE_STRUCTURE)) {
command = new TwineBallRetrieveCommand();
} else if (functionName.equals(NodeLevelOperations.RETRIEVE_ALL)) {
command = new TwineBallRetrieveAllCommand();
} else {
command = new TwineBallBaseCommand();
}
command.setObjectSerializer(objectSerializer);
command.setObjectNaming(objectNaming);
command.setMaxRecords(maxRecords);
command.setMetadata(metadata);

if (functionName == NodeLevelOperations.DELETE_NODE) {
command.setExecutionOrder(CommandForCursor.BEFORE_PARENT);
} else {
command.setExecutionOrder(CommandForCursor.AFTER_PARENT);
}

}catch (Exception e) {
throw new ResourceException(e);
}
return command;
}
```

Implementing Interaction.execute():

To enable the command pattern capability, you implement a class from the InteractionSpec with a call to the Command Manager. The InteractionSpec is part of the JCA CCI interface.

The interaction class is part of the JCA CCI interface and processes records as input and output. This is the API that is exposed for manipulating data in outbound operations:

```
public Record execute(InteractionSpec ispec, Record inRecord)
                                      throws ResourceException;
```

You want to call the Command Manager to produce the command structure, then the interpreter to execute each command in the structure. A simplified version of the resulting interaction code will resemble the following:

```
public Record execute(InteractionSpec ispec, Record inRecord) throws ResourceException {
WBIStructuredRecord wbiRecord = (WBIStructuredRecord) inRecord;
String functionName = ((WBIInteractionSpec) ispec).getFunctionName();
CommandForCursor topLevelCommand = commandManagerForCursor.produceCommands(wbiRecord, functionName);
interpreter.execute(topLevelCommand);
WBIStructuredRecord outRecord = new WBIStructuredRecord();
outRecord.setOperationName(functionName);
outRecord.setTwineBallConnection(connection.getEISConnection());
outRecord.setEISRepresentation(topLevelCommand.getEisRepresentation());
return outRecord;
}
```

Notice that you need not ″walk″ the incoming object structure, or the command structure– the command manager and interpreter perform this function.

Data and metadata

Adapter Foundation Classes (AFC) implement DESPI APIs and support two data formats, service data objects (SDO) and JavaBeans.

The data format-specific implementations are provided in the AFC and are abstracted from the adapters which use the DESPI APIs to process data in a format-independent way.

XSD and JavaBean structure relationship to DESPI

SDO Structure:

JavaBeanRecord Structure:

The JavaBeanRecord data structure is derived from the XML Schema defining the SDO. Adapter foundation classes provide an implementation of RecordGenerator which generates JavaBeanRecords for a given SDO/XML Schema.

The RecordGenerator class is used by the adapter enterprise metadata to generate the required artifacts for Java Bean Record data format. See Enterprise metadata implementation details on how to update the adapter enterprise metadata (EMD) implementation to use the RecordGenerator class.

Type mappings: Bean properties map to the attributes of a given complex type. The types of the simple attributes are derived from the XML built-in data types defined in the input schema for each subelement of the complex type. The sub-elements containing n-cardinality child objects are always defined as an array of the child type while single cardinality sub-elements are represented as a JavaBean property with the type of the subelement.

The following is a table showing the mapping between the built-in XML Schema Definition (XSD) types and bean properties generated by the RecordGenerator implementation in the AFC.

Table 2. Mapping between built-in XSD schema and JavaBean properties

Header	Header
Boolean	boolean
String	String
Int	int
Integer	Integer
Decimal	BigDecimal
Double	double
Float	float
Long	long
Short	short
hexBinary	byte[] (byte array)
Byte	byte
dateTime	Calendar
Date	Calendar
Time	Calendar

Table 2. Mapping between built-in XSD schema and JavaBean properties (continued)

Header	Header
anySimpleType	String
Any	String

There are cases in which a simple XML data type must be mapped to the corresponding Java wrapper class for the Java primitive type:

- an element declaration with the nillable attribute set to `True`
- an element declaration with the minOccurs attribute set to 0 (zero) and the maxOccurs attribute set to 1 (one) or absent
- an attribute declaration with the use attribute set to optional or absent and carrying neither the default nor the fixed attribute

The following shows examples of each:

- `<xsd:element name="code" type="xsd:int" nillable="true"/>`
- `<xsd:element name="code2" type="xsd:int" minOccurs=""></xsd:element>`
-
```
<xsd:element name="">
  <xsd:complexType>
  <xsd:sequence/>
  <xsd:attribute name="">
  </xsd:attribute></xsd:complexType>
 </xsd:element>
```

The element/attribute declarations for code, code2, code3 above are all mapped to the java.lang.Integer type.

JavaBean Metadata ASI format: The metadata is derived mostly from the structure of the bean. Other metadata, such as containedType and maxLength are not normal parts of a JavaBean structure, so they must be part of the annotation maps.

Annotation maps:

Annotations are not normally part of a JavaBean structure, so JavaBeanRecords must contain annotations in a specific format to be usable by the metadata API.

The JavaBeanRecord must implement the BeanMetadata interface.

```
public interface BeanMetadata {
 public  Map getObjectAnnotations();
 public  Map getPropertyAnnotations();
 public  Set getSetAttributes();
}
```

Each property (if it needs metadata) will have its own Map, which will be stored in the propertyAnnotations map, using the property name as key.

Reserved keys in the propertyAnnotations map:

- ContainedType

  The class of the object that this property contains
- PrimaryKey:

  Whether this property is key
- DefaultValue:

  The default value
- MaxLength:

  The maximum length of this property.

The notion of whether or not a property is set is critical to processing null values or values that have not been set in the adapter. If a property has been explicitly set to null, the adapter must be able to detect this and set the associated property to null in the EIS.

If the property has not been set, it might still have a null value, but the adapter will ignore the value and not set it in the EIS system.

A list of bean properties for which the setter is called is returned when getSetAttributes() is called. This will enable the DESPI implementation to determine if a particular attribute has been set.

Property order:

The notion of property order is critical, because the adapter may need to iterate over the properties in the same order that they appear in the EIS system.

Since JavaBean APIs cannot detect the order present in the class file, there must be a string array called ″propertyOrder″ in the bean, containing the names of all properties in the desired order.

```
public static final String[] propertyOrder = {"property1","property2"};
```

SDO to JavaBeanRecord ASI Mappings: Annotations from the SDO/XML schema are read and stored in a Map structure in the bean. Here is a description of how the RecordGenerator would populate annotation maps. For each element in the metadata annotation, the generator would create an entry in the Map with the name of the element as the key.

- If the element is a simple type with single cardinality then the value of this element is added to the Map.
- If the element is a simple type with n-cardinality, then a List is generated containing one or more values of this element.

  The List is then added to the annotation Map.
- If the element is a complex type with single cardinality then a Map is created containing the mappings for elements of this child complex type.

  The Map is then added to the annotation Map.
- If the element is a complex type of multiple cardinality then a List is created.

  The List would contain one or more Map entries where each Map contains the mapping for the elements of the child complex type.

Here is an example of how the object level metadata annotation would look like in an SDO schema:

```
<annotation>
<appinfo source="http://www.ibm.com/xmlns/prod/websphere/j2ca/sap/metadata">
<sapasi:sapBAPIBusinessObjectTypeMetadata xmlns:sapasi="http://www.ibm.com/xmlns/prod/websphere/j2ca/sap/metadata">
 <sapasi:Type>BAPI</sapasi:Type>
 <sapasi:Operation>
  <sapasi:MethodName>BAPI_CUSTOMER_CREATEFROMDATA1</sapasi:MethodName>
  <sapasi:Name>Create</sapasi:Name>
 </sapasi:Operation>
 <sapasi:Operation>
  <sapasi:MethodName>BAPI_CUSTOMER_CHANGEFROMDATA1</sapasi:MethodName>
  <sapasi:Name>Updatewithdelete</sapasi:Name>
 </sapasi:Operation>
</sapasi:sapBAPIBusinessObjectTypeMetadata>
</appinfo>
</annotation>
```

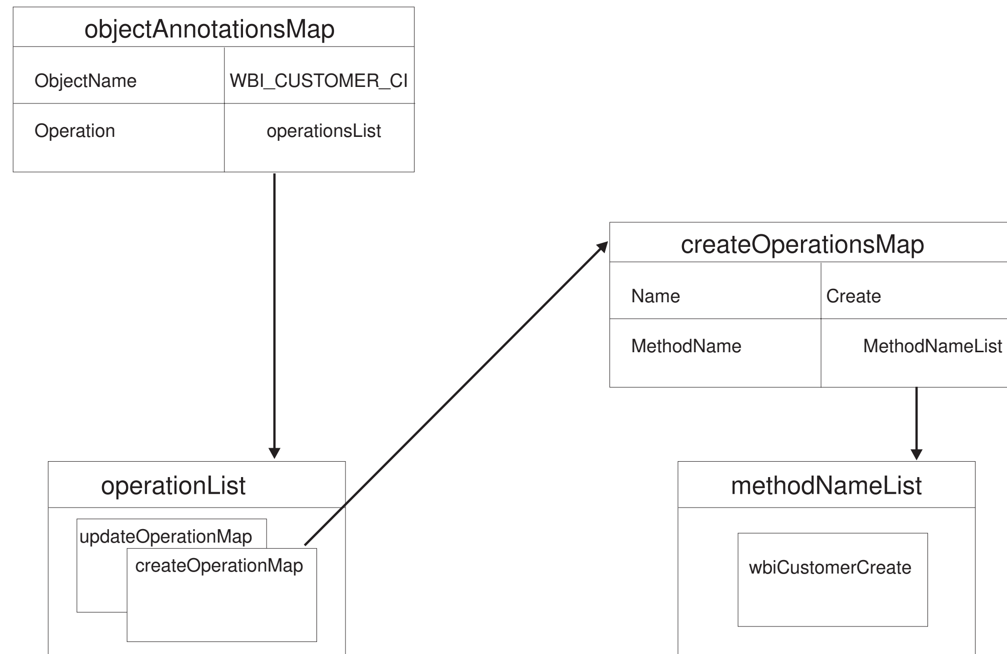
As defined in the sapBAPIBusinessObjectTypeMetadata schema ″Operation″ is an n-cardinality complex type. The ″MethodName″ element of the operation type is a simple type with multiple cardinality:

```
<complexType name="sapBAPIBusinessObjectTypeMetadata">
  <sequence>
   <element name="Type" type="string"/>
<element name="Operation" type="sapasi:sapBAPIOperationTypeMetadata" minOccurs="0" maxOccurs="unbounded"/>
<element name="ErrorConfiguration" type="sapasi:sapRFCErrorConfigurationMetadata" minOccurs="0" maxOccurs="1"/>
  </sequence>
 </complexType>

 <complexType name="sapBAPIOperationTypeMetadata">
  <sequence maxOccurs="1" minOccurs="0">
   <element name="Name" type="string" minOccurs="0" maxOccurs="1"/>
<element name="MethodName" minOccurs="0" maxOccurs="unbounded" type="string"/>
  </sequence>
 </complexType>
 <complexType name="sapRFCErrorConfigurationMetadata">
  <sequence maxOccurs="1" minOccurs="0">
<element name="ErrorParameter" type="string" minOccurs="0" maxOccurs="1"/>
<element name="ErrorCode" minOccurs="0" maxOccurs="1" type="string"/>
<element name="ErrorDetail" minOccurs="0" maxOccurs="1" type="string"/>
  </sequence>
 </complexType>
```

For the above metadata, the object level annotations map generated for the JavaBeanRecord would look like the following:

```
public static LinkedHashMap objectAnnotations = new LinkedHashMap();

static {
 objectAnnotations.put("Type","BAPI");

 LinkedList operationAnnotation = new LinkedList();

 LinkedList methodnameList;
 LinkedHashMap createOperationMap = new LinkedHashMap();

 createOperationMap.put("Name", "Create");
 methodnameList = new LinkedList();
 methodnameList.add("wbiCustomerCreate)";
 createOperationMap.put("MethodName", methodnameList);
 operationAnnotation.add(createOperationMap);

 LinkedHashMap updateOperationMap = new LinkedHashMap();
 updateOperationMap.put("Name", "Update");
 methodnameList = new LinkedList();
 methodnameList.add("wbiCustomerUpdate)";
 updateOperationMap.put("MethodName", methodnameList);
 operationAnnotation.add(updateOperationMap);

 objectAnnotations.put("Operation",operationAnnotation);
}
```

*ObjectAnnotations:*Object level metadata in the annotations is read and stored in the ′objectAnnotations′ Map. Following diagram shows the structure of objectAnnotations.



The Metadata API

Advanced implementations of adapters are metadata-driven. This implies that the adapter is not hard coded for each object type in the system, but rather has a form of discovery in which a representation of the object (the metadata) in the EIS is constructed at build time, then at runtime the adapter uses this metadata, along with the data, to process the object. Metadata can be in a preexisting format, such as a standardized schema. Standard metadata includes such information as property name, property type, maximum length, cardinality, and anything else that can be described in a standard schema. Metadata may also be in a format decided by the adapter. This form of metadata is called ″Application Specific Information″, or ASI. ASI can occur in three places.

- Object level metadata

  This includes the information about what type is being processed
- Operation level metadata

  This is context-specific object metadata, that is valid for the operation being processed at this time.
- Property level metadata

  This is information about the one property in the EIS schema. It may contain such information as the column name as it occurs in the EIS, which may be different than the property name in the object.

In order for the adapters to handle multiple representations of metadata; specifically SDO and JavaBean, the Foundation classes provide an API for the adapters and abstracts the metadata representation. The intention of this Metadata API is to present both structural and application-specific metadata to the adapter, so the adapter is insulated from the metadata format. For example, a JavaBean or an SDO may be used as metadata, but the adapter can use the same metadata APIs to walk over the structure and to extract the desired information from it.

Adapter implementations should use the following interfaces when retrieving metadata. Adapters should never cast to an implementation of these interfaces.

These interfaces may contain more helper methods than are listed here, please see the Javadoc for the additional helpers.

Factory classes

Class TypeFactory:

TypeFactory creates an instance of an implementation of Type. The TypeFactory is also capable of detecting whether SDO version 2, SDO version 1, or neither is present in the classpath, allowing it to make decisions about what implementation to use.

- `Type getType(Object object)`

  Gets a Type object from existing metadata.
- `Type getType(String namespace, String name)`

  Gets a Type object from a namespace, name combination. For a JavaBeans metadata implementation, the namespace is the package and the name is the class name.

Class SDOFactory:

SDOFactory creates an SDO instance, the implementation of which will depend on what version of SDO is in the classpath.

- `DataObject createDataObject(String namespace, String name)`

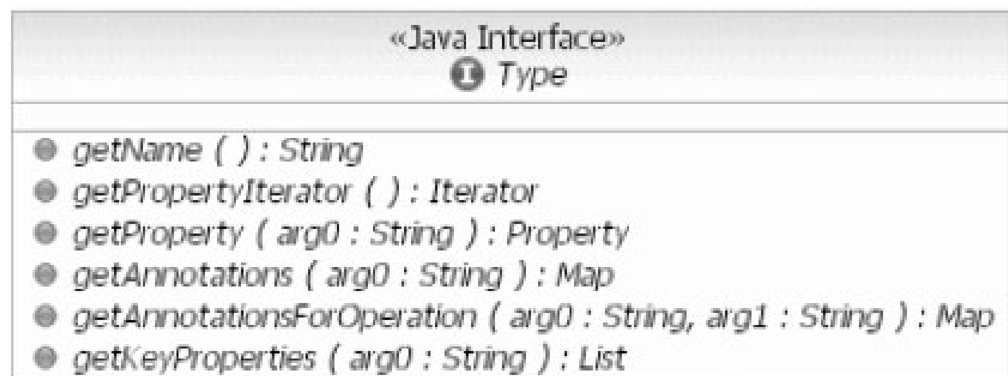
  Creates the data object based on the namespace and name.
- `DataObject createDataObject(Type type)`

  Creates the data object based on an instance of Type

Interfaces

Interface Type:

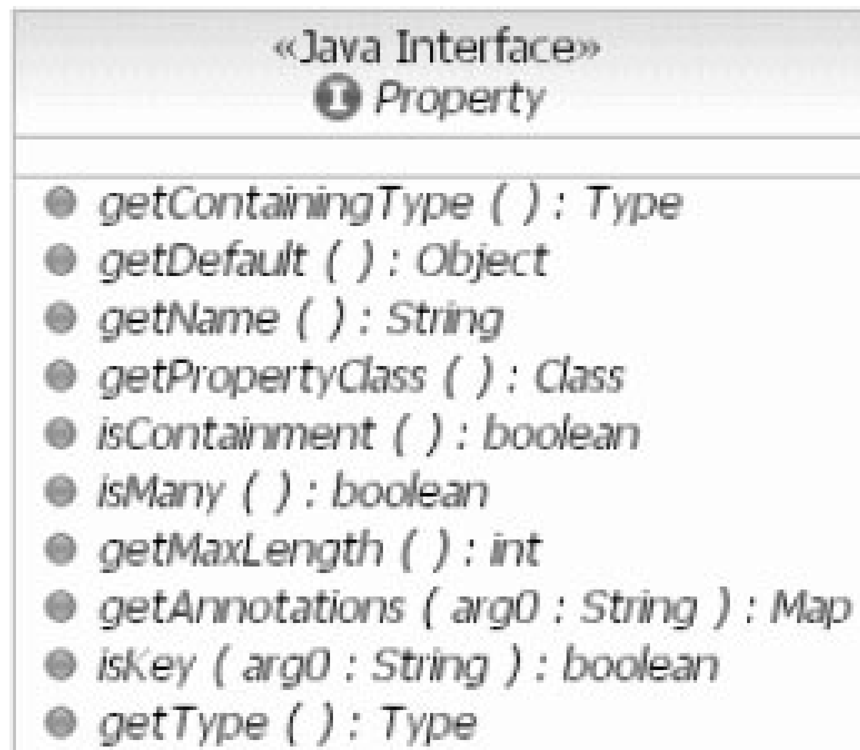
The interface type allows access to object-level metadata, including properties, object-level annotations and key properties.




```
String getName()
Retuns the name of the type.

Iterator getPropertyIterator()
Returns an iterator to allow iteraton over the properties in this type.

List getProperties()
Returns a list of properties for this type.
```

```
Property getProperty(String propertyName)
Returns the property object for this property name.

Map getAnnotations(String source)
Returns the object-level annotations for this type.  You must pass
in the "source" of the annotations, which corresponds to the "source"
attribute of the "appinfo" tag of the XML Schema representing this
object.  Annotations will be returned as a Map, and this Map may contain
other maps if the annotation structure is nested.

Map getAnnotationsForOperation(String source, String operation)
Often operation-specific object metadata is needed. E.g for
Create operation there is a specific sequence of APIs that have to
be executed, this set could be different for Update and Delete operations.
This method returns  the metadata for a given operation name as a Map of
name - value pairs.

List getKeyProperties(String source)
Returns the list of key properties in this type.
```

Interface Type:




```
Type getContainingType()
Returns the type containing this property.

Object getDefault()
Returns the default value for this property.

String getName()
Returns the name of this property.

Class getPropertyClass()
Returns the Class represented by this property.  For example, if the
property is of String type, this method will return String.class.
```

```
boolean isContainment()
Returns whether or not the property contains a Type
(complex object).

boolean isMany()
Returns whether or not the property contains a List or Array.

int getMaxLength()
Returns the max length of the property.

Map getAnnotations(String source)
Returns the annotations for this property.

boolean isKey(String source)
Returns true if this property is a key, and false if not.

Type getType()
If the property is containment, this method will return
the contained type.
```

Enterprise metadata implementation

Selection of artifact types

WebSphere adapters can run against multiple brokers (server runtimes).

Each broker might require different types of artifacts. The adapter foundation classes can generate artifacts in support of multiple brokers.

The following artifact types are supported by adapter foundation classes:

1. Data Bindings

   Data Binding classes generated by enterprise metadata discovery to support WebSphere Process Server runtime.
2. Generated Records

   JavaBean records generated from SDO to support clients that work with JavaBeans.
3. Generic Records

   Other DESPI implementations.

The table below provides a matrix for artifact type and their supported server runtimes.

Table 3. Artifact types and supported runtimes

	Runtimes		
Artifact types	**WebSphere Process Server**	**WebSphere Application Server**	**Other DESPI implementations**
DataBindings	X		
GeneratedRecords		X	
GenericRecords			X

While running the external service discovery wizard, adapter users can choose which artifact to generate depending on their runtime environment. Users can select more than one artifact.

Support for GeneratedRecords artifact type: WebSphere adapters may support JavaBeanRecord data representation along with SDO 1.0 and SDO 2.0 data objects. As part of the support for JavaBean data representation, the adapter foundation classes provides a JavaBeans record generator class that can generate DataBindingDescriptions for the object types discovered and selected through the enterprise metadata discovery (EMD) process.

Each adapter enterprise metadata discovery provides a list of artifact types it supports, allowing the user to select an artifact type that is appropriate for their environment. For example, to enable the adapters on runtimes where SDO implementations are not available, users will run the ESD wizards to generate the adapter artifacts for GeneratedRecords type. When the user selects a Generated Records artifact type in the enterprise metadata discovery process, the external service discovery wizard will look for a databinding generator class name on the EMD DataDescription and invokes that class to generate the JavaBeans records.

All adapter enterprise metadata discovery that support the generated records artifact type would need to set the data binding generator class name (on the DataDescription) to the generator implementation provided in the AFC. Here is code snippet showing how the record generator is set in the service description implementation of an adapter EMD.

```
WBISingleValuedPropertyImpl property = (WBISingleValuedPropertyImpl) selectionProperties.getProperty
(EMDConstants.ARTIFACTS_SUPPORTED);
WBIMetadataConnectionImpl.getToolContext().getProgressMonitor().setNote("Business object definitions created");

String namespace = getNameSpace();
//Change made for making BG Optional
if (property.getValue().equals(EMDConstants.DATA_BINDINGS)) {
 dataDescription.setName(WBIDataDescriptionImpl.convertNamespaceToUri(namespace + "/"
  + metadataObj.getBOName().toLowerCase() + "bg" ), metadataObj.getBOName() + "BG");
dataDescription.setDataBindingGeneratorClassName
("com.ibm.j2ca.sample.twineball.emd.runtime.TwineBallDataBindingGenerator");
dataDescription.setGenericDataBindingClassName(null);
}
else if (property.getValue().equals(EMDConstants.GENERATED_RECORDS)) {
  dataDescription.setName(WBIDataDescriptionImpl.convertNamespaceToUri(namespace + "/"
  + metadataObj.getBOName().toLowerCase()), metadataObj.getBOName());
 dataDescription.setDataBindingGeneratorClassName
  ("com.ibm.j2ca.extension.dataexchange.bean.generator.RecordGenerator");
            dataDescription.setGenericDataBindingClassName(null);
} else {
             //Generic Record Scenario
            dataDescription.setName(WBIDataDescriptionImpl.convertNamespaceToUri(namespace + "/"
            + metadataObj.getBOName().toLowerCase()), metadataObj.getBOName());
 dataDescription.setGenericDataBindingClassName
("com.ibm.j2ca.sample.twineball.TwineBallStructuredRecord");
```

New property types supported from WebSphere Adapter Toolkit V6.1:

TableProperty: A property representing a table with rows and columns. Each column is represented by the PropertyDescriptor instance and each cell corresponding to a given row and column is represented by a SingleValuedProperty implementation.

TreeProperty: A property representing a tree of selectable nodes. Each node is represented by a NodeProperty implementation which can be selected, highlighted and can have configuration properties represented by a PropertyGroup instance.

Enterprise Metadata Discovery general interfaces and implementation for application adapters

Enterprise metadata discovery is a *discovery service*, or a component within an adapter that enables the generation of business object definitions and other artifacts required by service component architecture.

The enterprise metadata discovery component is analogous to the Object Discovery Agent of WebSphere Business Integration Adapters. In addition to generating business object definitions, however, the enterprise metadata discovery also generates service component architecture artifacts such as an Import/Export file and WSDL. An explicit goal of enterprise metadata discovery is to enable existing JCA resource adapter extensions to provide metadata discovery and import in a simple, straightforward way.

Using adapters that allow for metadata discovery and import, you can create and edit services with the following capabilities, where the operation style is supported by the underlying adapter:

- Integration-framework-initiated operations to retrieve data from or modify data in the EIS.
- EIS-initiated operations, where the request originates within the EIS.

  Note: This type of operation is used for retrieving data from, or modifying data in, the integration framework.

Types of enterprise metadata

The enterprise metadata discovery service is responsible for exposing two categories of metadata: EIS metadata and service metadata

EIS metadata:

EIS metadata describes the system capabilities and business object structures employed by the EIS. The enterprise metadata discovery service acquires EIS metadata from the communication configuration information that characterizes system function and object interactions.

System Capabilities

At the system level the metadata describes the types of information managed by the EIS system. This is represented as a collection of functions on the EIS client interface or as a set of business objects that are accessed from the EIS client interface. In addition to function and object descriptions, EIS metadata includes a description of the styles of interaction that can occur with functions and objects in the EIS metadata model. There are two interaction styles:

- Inbound - EIS-initiated
- Outbound - Client-initiated

There can be two kinds of interaction for each interaction style.

- Request-Response - This kind of interaction takes a request and returns a response. Outbound interactions are of this type.
- One-way - A one-way interaction takes a request, but does not return a response. Inbound interaction style is one-way

The discovery service defines whether an interaction style is inbound versus outbound. The distinction between one-way and request-response is made by the method descriptions generated for the service. The request-response mode would have both input and output specified for a method description whereas input arguments only would be specified for one-way. The metadata generated by the discovery service must comply with the restrictions on interaction mode mentioned above: for Outbound the metadata must support Request/Response and for Inbound, one-way.

Business Object Structures

These are the business objects used by JCA adapters. They describe the structure and content of arguments to functions on the EIS client interface or the business objects accessed through the EIS client interface.

Communication Configuration

The communications configuration is represented by a collection of properties also referred to as PropertyGroup. The properties include properties required for both inbound and outbound communication. In some cases, there may be more than one set of properties required. These might be required to describe different types of outbound connections that for design time and runtime or for inbound connection configuration.

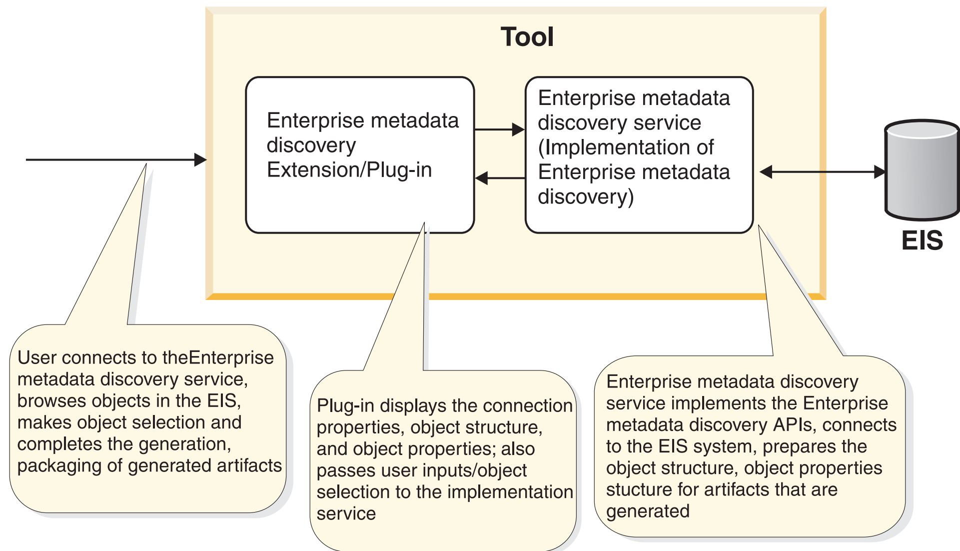
The discovery service is not limited to accessing metadata from the EIS. If the metadata is stored in a separate repository, the discovery service can mine the metadata from there. This allows the discovery service to combine all metadata at its disposal to provide a meaningful user experience to the tool user.

Service metadata:

Service metadata is generated by the adapter as business objects are imported into a service description.

Service metadata includes the service name and the adapter and managed connection factory properties that describe the configuration for an outbound connection to the EIS (outbound case). For an inbound connection to the EIS (inbound case), service metadata might include resource adapter and Activation Specification configuration properties. Service metadata also includes method-level metadata such as the method name, data description for the input and output, and either InteractionSpec properties (outbound case) or the EIS function name (inbound case).

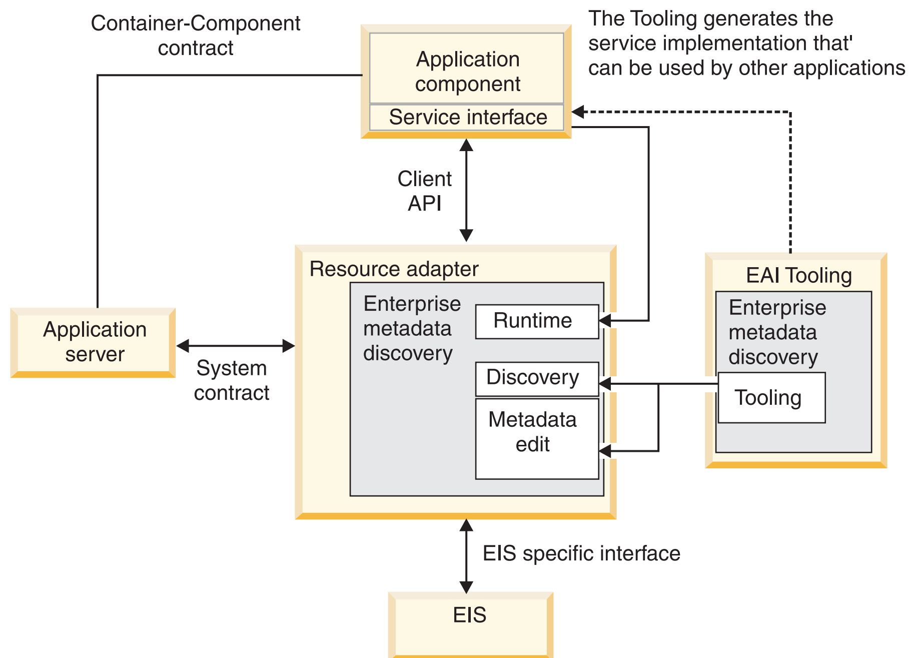
For most cases, service metadata representing outbound and inbound connections is the same as the EIS metadata for communication configuration. For other cases, depending on specific implementation, the service metadata might be distinctly different. For example, the discovery service might find objects from a different interface, such as a separate repository, or might use a different type of a connection for discovery.



Service metadata

Enterprise metadata discovery architecture

The enterprise metadata discovery tooling includes runtime, discovery, and service generation interfaces and metadata edit capabilities.



Enterprise metadata discovery architecture

Note: The solid arrows represent the enterprise metadata discovery implementation in the above diagram.

The components of the enterprise metadata discovery implementation, described below, are:

- **Runtime**

These interfaces extend the CCI interfaces defined in the JCA specification to support invocation of services discovered with enterprise metadata discovery. These interface implementations are provided by the resource adapter provider or a third party discovery service provider. These are the data binding interfaces that are used to integrate with service data objects.

- **Discovery**

  These interfaces constitute the bulk of the contracts defined in this specification. They define the following contracts:

  - Discovery - These interfaces allow the tool to browse EIS metadata.
  - Connections - These interfaces allow the tool to discover all the available connections, create connections, edit connections and persist the connections for future use.
  - Mutable metadata - These interfaces allow the tool to edit the EIS metadata that is being browsed.

- **Service Generation**

  These interfaces allow the tool to create or edit service descriptions that define the service a user has selected. These service descriptions are used to generate service implementations that are deployable in application servers and used as part of a service oriented architecture. These interfaces also provide support for editing application-specific information schemas.

  These interface implementations are provided by the resource adapter provider or a third party discovery service provider.

- **Metadata edit**

  This interface allows the EAI tool to edit the configuration of a resource adapter or a service. This interface implementation is provided by the resource adapter provider or a third party discovery service provider.

Metadata discovery

This primary interface to the enterprise metadata discovery plug-in holds a reference to the ToolContext. ToolContext is used for logging and tracing and to provide information that helps perform discovery.

The discovery service must specify the following for this interface.

1. List of AdapterType - For JCA adapters only one AdapterType comprises this list.
2. MetadataTree - Represents the object tree structure that would be displayed by the tool. Each node in the tree is represented by an instance of a MetadataObject. The nodes can have their own properties. The tool can also apply filter properties when fetching children for the node.
3. Service Description - Created on request by the user for an inbound or outbound service. Includes all of the objects selected prior to a request.

Metadata discovery adapter type

Adapter type metadata identify aspects of the adapter that are supported by the enterprise metadata discovery implementation. The Adapter Foundation Classes provide an interface for adapter type information.

The Adapter Foundation Classes provide an implementation for this interface: `WBIAdapterTypeImpl`. Individual discovery service implementations should use this base class and not implement their own.

Multiple `adapterTypes` can be supported by a single enterprise metadata discovery implementation. A use case might involve a single adapter instance as a channel

for multiple back-end EIS assets. IBM WebSphere recommends a single adapterType for each enterprise metadata discovery implementation.

The following information must be provided for this class by a discovery service implementation.

- ID (for example, PeopleSoft)
- Description (for example, PeopleSoft JCA Adapter)
- Display name (for example, PeopleSoft Adapter)
- Vendor (for example, IBM)
- Version (for example, 6.2)
- List of OutboundConnectionTypes - Lists the outbound connection types an adapter can support. A single adapter might support multiple connection types. The mapping of connection type to adapter is driven by the number of managed connection factories that are supported by the adapter. If the adapter uses a different connection than one provided by a managed connection factory, this list must include the additional connection types. The connection types displayed in the tool for selection are only the ones that can be used to perform metadata discovery. Such connections are marked as `true` for `isSupportedInMetadataService`. For all connection types that cannot be used to perform discovery, set `isSupportedInMetadataService` to `false`.
- List of InboundConnectionTypes - List of inbound connection types supported by an adapter. The inbound connections map to the Activation Specifications supported by an adapter.

Metadata discovery connection type

You must specify connection type values for the enterprise metadata discovery service. The connection type includes connection configuration instance information for outbound and inbound directions.

ConnectionType is a factory for ConnectionConfiguration instances. The enterprise metadata discovery tool uses the instances to create actual and persistent connections. The connection can be inbound or outbound and is unique within the AdapterType.

OutboundConnectionType

The enterprise metadata discovery service uses the OutboundConnectionType property to create an outbound service description. The mapping of connection type to adapter is driven by the number of managed connection factories that are supported by the adapter. If the adapter uses a different connection than one provided by a managed connection factory, this list must include the additional connection types. The connection types displayed in the tool for selection are only the ones that can be used to perform metadata discovery. Such connections are marked as `true` for `isSupportedInMetadataDiscovery`. For all connection types that cannot be used to perform discovery, set `isSupportedInMetadataDiscovery` to `false`.

IBM WebSphere recommends implementing a connectionType for metadata that is separate from that for run time. The metadata `connectionType` must contain properties needed to perform the metadata discovery only. The runtime connection type should have properties for the resource adapter and the managed connection factory. The discovery service should attempt to reuse the properties from the metadata `connectionType` wherever possible. Candidates for reuse include the username and password properties. A utility class, `EMDUtil`, provides methods to

copy matching properties from metadata connection to runtime connection. For more information, see the `WBIOutboundConnectionTypeImpl` in the Javadocs.

The following information must be provided for this class by a discovery service implementation:
- ID (for example, PeopleSoft)
- Display name (for example, PeopleSoft)
- Description (for example, PeopleSoft Component Interface)
- `ResourceAdapterBean` class (for example, ″com.ibm.j2ca.peoplesoft.PeopleSoftResourceAdapter″)
- `ManagedConnectionFactory` bean class (for example, ″com.ibm.j2ca.peoplesoft.PeopleSoftManagedConnectionFactory″)
- `OutboundConnectionConfiguration` - Represents properties needed to connect to the EIS for discovery and to create service descriptions. For connection types that are used only at runtime this represents managed connection factor and resource adapter properties.
- `isSupportedForMetadataDiscovery` - is set to true if the `connectionType` can be used for metadata discovery. Only valid, specified connection types are displayed for `connectionType` selection.

InboundConnectionType

InboundConnectionType is used to create Inbound service description. An adapter can support multiple Inbound connection types, the mapping of inbound connection is to Activation Specs, as many activation specs an adapter supports than many inbound connection types would have to be supported.

The following information has to be provided by discovery service implementation for this component.
- ID (for example, PeopleSoft)
- Display name (for example, PeopleSoft)
- Description (for example, PeopleSoft Component Interface)
- `ResourceAdapterBean` class (for example, ″com.ibm.j2ca.peoplesoft.PeopleSoftResourceAdapter″)
- `ActivationSpecBean` class e.g. ″com.ibm.j2ca.peoplesoft.PeopleSoftActivationSpec″
- `InboundConnectionConfiguration` - Represents resource adapter and Activation Specification bean properties.

Discovering System Capabilities

After selecting the ConnectionType, enter the properties for connection configuration. This typically includes those properties needed to connect to the EIS or EIS repository and also properties required to define the artifacts that will be generated by enterprise metadata discovery. Once you enter the properties the tool requests, the enterprise metadata discovery service attempts to establish a connection to the EIS repository. The service extracts values for configuration properties and opens a connection to the EIS repository. This is represented by MetadataConnection object.

MetadataConnection

The MetadataConnection object represents the connection to EIS or EIS repository. This interface is implemented by the Adapter Foundation Classes only and does not require any implementation from discovery service instances.

MetadataConnection uses the managed connection factory to create the connection to the EIS. The configuration properties specified by the user are mapped to the managed connection factory. AgetConnection() call is placed to obtain an EIS connection.

Besides the physical connection it represents, MetadataConnection also has a handle to the ConnectionType and OutboundConnectionConfiguration.

Once the connection is established, enterprise metadata discovery makes the request to get the MetadataTree from the MetadataDiscovery implementation. If there is a filter PropertyGroup specified at the MetadataTree you are prompted enter those properties.

MetadataTree model:

The enterprise metadata discovery tool displays a MetadataTree model that represents an object structure in the EIS. The display also reflects your enterprise metadata discovery service implementation and filters that you can apply to the MetadataTree model to improve usability.

The structure of the MetadataTree reflects not only the EIS business data but also your enterprise metadata discovery implementation. For example, one implementation might display the properties of the nodes or MetadataObject in the MetadataTree model. Another implementation might display function parameters as nodes in the MetadataTree model.

IBM WebSphere recommends displaying the leaf-level properties in the model only if there is an advantage to doing so. In most cases, simple properties or function parameters should not be added as nodes in the tree. The description of the node representing the object or the function should provide information about the node. For example, in an EIS where function overloading is possible, the function description for the node should show enough parameter information to make the right import selection.

To improve usability, provide filters on MetadataTree models as applicable. For example, if the EIS contains many objects and is difficult to search, a filter might help. A best practice recommendation is to provide an option of inbound or outbound as a filter property for the MetadataTree. Doing so restricts the objects selected to one type for each discovery service pass. Where the same objects are used for inbound and outbound, skip this property and define it after choosing objects for MetadataSelection.

For more information, see the WBIMetadataTreeImpl in Javadocs.

MetadataSelection:

MetadataSelection is a container that holds objects and properties that you select to guide or filter enterprise metadata discovery.

In addition to object selection, `MetadataSelection` also holds properties that are applicable for the selected objects. Such properties include the following:

- Specification of service type: inbound or outbound
- Namespace for use by business object definitions
- `ConnectionType` that for use at run time.

  This step is required if the `ConnectionType` at run time differs from that for metadata discovery. Regardless, IBM WebSphere recommends a different `connectionType` for metadata than for run time. The metadata `connectionType` should contain properties needed to perform the discovery only. The runtime `connectionType` should contain resource adapter, managed connection factory, or `ActivationSpecWithXid` properties.

Implement the discovery service to reuse properties (for example, username, password) from the `MetadataSelection` `connectionType` wherever possible. You can specify other properties after the service description is created, when the tool checks whether artifact properties have been specified.

MetadataObject:

The `MetadataObject` represents the node of the `MetadataTree`. You define properties and filters in this object that help guide enterprise metadata discovery.

IBM WebSphere recommends that you specify `objectProperties` that suggest the EIS objects you are defining. You should also define filter properties in `MetadataObject` that might make fetching of child objects more efficient and usable. For example, for JDBC, if the node represents a schema, you might define a filter to fetch tables by name rather than fetching all tables.

Enterprise metadata discovery description APIs

The enterprise metadata discovery description APIs comprise the objects that correspond to the artifacts generated by the discovery process. The artifacts include business object XSDs, the SCA import and export files and WSDL files.

The description APIs embody the service metadata generated by the discovery service in response to import requests. The high-level model for this information includes the following:

- `ServiceDescription`
- `FunctionDescription`
- `DataDescription`
- `ConnectionDescription`

Service, function, and data descriptions are present for both inbound and outbound services. The models for the connection and function descriptions differ slightly for inbound and outbound (for consistency with the JCA 1.5 specification). The data description is identical for both inbound and outbound.

Service descriptions:

You extend abstract service description classes and implement methods to define a service description for an inbound or outbound object.

You can specify a service description for inbound or outbound connections. The objects you select are maintained by the `MetadataSelection` object. The `MetadataSelection` is passed to the discovery service implementation to complete

the import. The Adapter Foundation Classes provide an implementation for ServiceDescription as abstract classes. Discovery service implementations should extend these classes and implement the abstract methods

Inbound service descriptions

Enterprise metadata discovery service implementations should provide the following information for an inbound service description.

- Service name - The implementation must provide a default service name, (for example, PeopleSoftInboundService), but you can specify this property.
- Inbound connection description - This is the connection description for inbound, typically resource adapter and Activation Spec beans
- Function list - These represent the methods and functions that can be invoked through the inbound connection. An inbound function description describes the interface for each of these methods.
- EIS function selector - A function selector is a class provided by the adapter developer or the integration developer that takes as input the message received from the adapter and returns the name of the EIS function that the message represents. This is used by the SCA run time to determine which method to invoke on the next SCA component. The JCA adapters would use the base class implementation of FunctionSelector. The default value for this property is the name of Generic Function selector provided by the Adapter Foundation Classes. The discovery service can override the default value.

Function selector

The function selector provides a unique name for the EIS function name. Due to SCA restrictions, method overloading cannot be used in interface definitions. Accordingly, you cannot have two methods with same name but which accept different arguments. For most inbound descriptions, the method name and the EIS function name are identical.

The base class FunctionSelector creates the EIS function name based on whether the input object is an AfterImage or a Delta. If a delta, then the name returned is emitDelta_*business_object_name*; if an AfterImage, then name returned is emit_*TopLevelVerb*_AfterImage_*busineess_object_name*.

See Business object names for more information about naming business objects.

Outbound service descriptions

Discovery service implementations must provide the following information for outbound service descriptions.

- Service name - The implementation must provide a default service name.
- Outbound connection description - his is the connection description for outbound, typically resource adapter and Activation Spec beans
- Function list - These represent the methods and functions that can be called by the client. An Outbound Function Description describes the interface for each of these methods.

Connection descriptions:

You must provide EIS connection descriptions for the enterprise metadata discovery service. Adapter Foundation Classes contain interface implementations to help you get started.

Like `ServiceDescription`, `ConnectionDescription` can be either inbound or outbound service. Adapter Foundation Classes provide implementation for `ConnectionDescription` interfaces. Discovery service implementations need not implement these interfaces.

Inbound Connection Description

The following information must supplied by the discovery service implementations:

- `ResourceAdapter` - This is an instance of the `ResourceAdapter` JavaBean that will be used to configure a resource adapter instance.
- `ActivationSpecWithXid` - This is an `ActivationSpecWithXid` instance that will be passed to `ResourceAdapter.endpointActivation()` at run time to support the methods defined on the inbound service.

Outbound Connection Description

The following information must supplied by the discovery service implementations

- `ResourceAdapter` - This is an instance of the `ResourceAdapter` JavaBean that will be used to configure a resource adapter instance.
- `ManagedConnectionFactory` - This is an instance of the `ManagedConnectionFactory` JavaBean that will be used to configure an instance of the `ManagedConnectionFactory` at run time for creating outbound connections.

Function descriptions:

Function descriptions apply to either inbound or outbound service. The Adapter Foundation Classes provide implementations for the `FunctionDescription` interfaces. Discovery service implementations need not implement these interfaces.

Inbound Function Description

The following information has to be filled in by the discovery service implementations:

- Name - e.g. `emitDeltaCustomer` or `emitCreateAfterImageCustomer`
- EIS Function Name - the name of the function that would be returned by function selector. E.g. `emitDeltaCustomer` or `emitCreateAfterImageCustomer`.
- Input - input data description representing the object that is input to the method
- Output - output data description representing the object that is returned by the method. This would be null for JCA adapters as they support only One-way interaction style.
- MetadataImportConfiguration - Handle to the metadata import configuration or the object that was selected by the user which lead to creation of this method description.

See "Inbound Operation Signatures" in Standard operations for descriptions of the inbound operation signatures and scenarios for their use.

Outbound Function Description

The following information has to be filled in by the discovery service implementations:

- Name - e.g. createCustomer, applyChangesCustomer
- InteractionSpec - Instance of interaction spec which has function name specified that represents this method description. E.g. for createCustomer the function name would be 'Create'.
- Input - input data description representing the object that is input to the method
- Output - output data description representing the object that is returned by the method.
- MetadataImportConfiguration - Handle to the metadata import configuration or the object that was selected by the user which lead to creation of this method description.

See "Outbound Operation Signatures" in Standard operations for descriptions of the outbound operation signature and scenarios for their use.

Data descriptions:

The data description implementation enables the adapter to create valid data objects for EIS requests and to interpret the objects returned as responses. You must implement `DataDescription`, `InboundServiceDescription`, and `OutboundServiceDescription`.

DataDescription

The data description is common to inbound and outbound service. It includes a definition of the structure and content of adapter business objects that will be passed between the client and adapter at run time. Each `DataDescription` instance should have a unique namespace. The convention followed by IBM is to use a base namespace concatenated by the name of the corresponding object.

InboundServiceDescription

The `InboundServiceDescription` must have a default name and associated function descriptions. The standard top-level operations are described in Inbound Operation Signatures. InboundFunctionDescription should use the same functionName and EISFunctionName.

Note: If needed, the default FunctionSelector can be overridden.

OutboundServiceDescription

The OutboundServiceDescription must have a default name, along with associated function descriptions. The supported, standard top-level operations are described in the Outbound operation signatures table in Standard operations.

Adapters that can provide functionality above and beyond these standard operations can opt to expose unique, EIS-specific operations. giving customers access to more functionality of the EIS. Define names and behavior on a case-by-cases basis to most accurately reflect the functionality of the underlying EIS.

Note: To limit confusion, custom operation names should not conflict with the standard operation names mentioned above.

Note: In cases where the mapping of EIS operations to Create, Retrieve, Update and Delete is equivalent except for naming conventions–for example, PeopleSoft has a Find operation that is functionally equivalent to a Retrieve–the EIS-specific operation should not be exposed since it adds no value.

Note: Function-based adapters occasionally may be required to perform multiple individual EIS operations to achieve the equivalent of a single Create, Retrieve, Update, or Delete operation. IBM WebSphere recommends that these low-level EIS operations also be exposed so that users are free to compose equivalent operations.

Business object structures for enterprise metadata discovery

In the Adapter Foundation Classes, `BusinessObject` and its `BusinessObjectAttribute` and `Metadata` structures facilitate the generation of XML schema definitions.

The Adapter Foundation Classes allow discovery service implementations to use Java APIs to prepare the `BusinessObject` structure. This `BusinessObject` structure can then be serialized to create an XML schema document string. This will reduce the effort for individual discovery service implementations, as they do not use XML parsers or general string concatenation functions to prepare the XML schema definitions.

BusinessObject:

BusinessObject contains properties and methods that help the discovery service generate object definitions.

The discovery service must provide the following for `BusinessObject`:

- `Name` – For example, PURCHASEORDER
- `BusinessObjectAttributes`
- `Namespaces` – These name spaces are added to the XSD definition.
- `TargetNameSpace` – For example, http://www.ibm.com/xmlns/prod/websphere/j2ca/peoplesoft/purchaseorder.
- `ImportedNameSpaces` – These name spaces would be added as `<import..` in XSD.
- `IncludeNameSpaces` – These name spaces would be added as `<include..` in XSD.
- `TopLevel` – Boolean if business object is `TopLevel` and needs a business graph definition
- `Metadata` – Metadata object representing the business object level application-specific information.

`BusinessObject` provides two methods:

- `Serialize` – To obtain the XSD string for the object.
- `getGraphDefinition` – To obtain the business graph definition for a top-level business object.

BusinessObjectAttribute:

BusinessObjectAttribute provides information that further defines the business object for the discovery service.

The discovery service must provide the following for `BusinessObjectAttribute`:

- `Name` – For example, VENDORID
- `Type` – For example, `string`
- `Cardinality` – 1 or *N*
- `Required` – Boolean
- `ObjectTypeName` – For cases where an attribute maps to an object for example, porecord:PORECORD.
- `Max Length`
- `Metadata` – the `MetadataObject` representing attribute-level application-specific information.

ImportedNamespace:

ImportedNamespace describes the names that must be imported into the business object definition.

This class represents the `import` statement in the XML schema definition. There are two properties that discovery service must provide:

- NameSpace – The name that must be imported
- Location – The location of the file that must be imported

Metadata:

The `metadata` component describes information about a business object definition and its attributes.

The business object and attribute level metadata should be modeled using this component. The discovery service must provide the following information for this object:

- `Source` – Added in the application-specific information statement. For example, WBI.
- `NameSpace` – Added to application-specific information clause.
- Name-value pairs for application-specific information properties (HashMap). – For example, `pasi:Getter value getObjectName`.

Namespace definition

The namespace http://www.ibm.com/xmlns/prod/websphere/j2ca is reserved for XML artifacts produced by or employed by WebSphere Adapters.

Adapters should use `http://www.ibm.com/xmlns/prod/websphere/j2ca/<adapter_name>` for any adapter-specific artifacts generated during the metadata import process; for example, `http://www.ibm.com/xmlns/prod/websphere/j2ca/peoplesoft`.

For Business Objects the name of the business object should be appended to the base name space; for example,`http://www.ibm.com/xmlns/prod/websphere/j2ca/peoplesoft/purchaseorder`.

For ASI schema the target namespace used must be `http://www.ibm.com/xmlns/prod/websphere/j2ca/peoplesoft/metadata`.

Implementing enterprise metadata discovery classes

The following subsections describe the steps and constructs required for implementing an enterprise metadata discovery instance.

BootStrap:

WebSphere Integration Developer performs a bootstrap step to identify a resource adapter that has been enabled for enterprise metadata discovery service.

To identify a resource adapter that is enabled for metadata discovery service, WebSphere Integration Developer launches a bootstrap step to find a `discovery-service.xml` file. This file should be located in the **META-INF** folder for the resource adapter. An example of this file is found in the sample delivered with the WebSphere Adapter Toolkit.

Resource bundles:

Resource bundle files capture properties and logging and tracing messages that are germane to an enterprise metadata discovery implementation.

Each enterprise metadata discovery implementation must have two resource bundle files. One captures the property group and property names and descriptions for properties displayed by WebSphere Integration Developer tooling. The other is for messages written to the log file for tracing. The bundle names must follow the convention of including the `EMD.properties` file in the enterprise metadata discovery package as shown in the following examples:

- `com.`*<company name>*`.j2c.<app name>.emd.EMD.properties` to capture resource bundles for property groups
- `com.`*<company name>*`.j2c.<app name>.emd.LogMessages.properties` to capture logging and tracing resource bundles.

The property groups representing resource adapter, managed connection factory, and the Activation Specification must have property names matching their bean properties.

Property groups:

All the properties used in the discovery service are represented by the `PropertyGroup` set of interfaces.

A property group is a collection of properties. A property group can be associated with the `Inbound`- and `OutboundConnectionConfiguration`, `MetadataTree`, nodes of the `MetadataTree` (`MetadataObject` and `MetadataSelection`). `PropertyGroup` supports nesting and can therefore include child `PropertyGroups`. It also provides a listener and an event interface to trickle changes from one property into another property or property group.

The Adapter Foundation Classes provide a complete implementation of the `PropertyGroup`APIs. Individual discovery service implementations should not implement these APIs.

The keys properties of this API set are as follows:

- `SingleValuedProperty` – Allows a single property of any Java type. It includes attributes such as required, sensitive, hidden, primitive, default value, and valid values.
- `MultiValuedProperty` – Allows a property to be represented as a list of values. For example, you can represent the operations for a `BusinessObject` using this type of property and then make multiple selections.

- PropertyGroup – A collection of properties including single and multi types and PropertyGroup itself. For example, OutboundConnectionConfiguration allows three property groups in one Main property group: UserConfiguration that includes Username and Password; MachineConfiguration that includes Hostname and PortNo; and Miscellaneous that includes other properties such as Prefix and DirectoryName.
- PropertyChangeListener – Used when a property change affects some other property or property group. This can be associated to any property or a property group. Each property can, in turn, have associated listeners that it can notify when a change happens. For example, Property A default value must change when Property B's value changes. Accordingly, the PropertyChangeListener that references Property A will be added to the listener list for Property B. When property B is changed in the set method, propertyChange can be fired on all listeners in the list. This will lead to changes in Property A.
- PropertyChangeEvent – An event that would be passed when propertyChange is fired on PropertyChangeListener. It includes the type of change, such as PROPERTY_VALUE_CHANGE, PROPERTY_ENABLED, PROPERTY_DISABLED, PROPERTY_VALID_VALUES_CHANGE, OLD_VALUE, NEW_VALUE, and SOURCE_OF_CHANGE.
- TableProperty A property representing a table with rows and columns. Each column is represented by the PropertyDescriptor instance and each cell corresponding to a given row and column is represented by a SingleValuedProperty implementation.
- TreeProperty A property representing a tree of selectable nodes. Each node is represented by a NodeProperty implementation which can be selected, highlighted and can have configuration properties represented by a PropertyGroup instance.

Enterprise metadata discovery implementation samples

The code samples in this section are from the TwineBall sample enterprise metadata discovery implementation.

For the precise class structure and additional information, refer to the code for the TwineBall enterprise metadata discovery implementation that is delivered with the WebSphere Adapter Toolkit.

Logging and Tracing:

This describes the logging and tracing implementation for enterprise metadata discovery.

Use the WBIMetadataDiscoveryImpl.getLogUtils() call to acquire an LogUtils instance. Then use the appropriate method to perform logging and tracing.

Property group sample:

Use the property group APIs to create property groups required for an enterprise metadata discovery implementation.

To enable validation of specific properties, extend the WBISingleValuedPropertyImpl or WBIMultiValuedPropertyImpl and then implement the vetableChange() method. Any validation can be performed in this code. In case of failures, PropertyVetoException must be thrown.

A propertyChange() method should be implemented if a property needs to listen for changes on some other property. Check the TwineBall sample for ServiceTypeSingleProperty.

To enable the function of both vetoableChange and propertyChange, the instance of the property should be added to the propertyChangeListener list. In the TwineBallMetadataSelection class in the TwineBall sample, the property representing operations listens on the property for serviceType:

```
propertyGroup = new WBIPropertyGroupImpl(Constants.SELECTION_PROPERTIES);

propertyGroup.setDisplayName(WBIMetadataDiscoveryImpl.getPropertyName
                                (Constants.SELECTIONPROPERTIES));
propertyGroup.setDescription(WBIMetadataDiscoveryImpl.getPropertyDescription
                                       (Constants.SELECTIONPROPERTIES));
WBISingleValuedPropertyImpl typeProp = createServiceTypeProperty(propertyGroup);
ServiceTypeSingleProperty operationProp = createNamespaceProperty(propertyGroup);
createMaxRecordProperty(propertyGroup);
createRelativePathProperty(propertyGroup);

// Copy the applied properties to the new instance
if (this.getAppliedSelectionProperties() != null)
       EMDUtil.copyValues(this.getAppliedSelectionProperties(), propertyGroup);
typeProp.addPropertyChangeListener(operationProp);
```

Progress monitor sample:

For enterprise metadata discovery processes that are time-consuming–such as retrieving information from an EIS and building MetadataObject instances–you can use the progress monitor.

While you are running the discovery service, the progress monitor can capture information on current processes and allow you to cancel operations. You can locate a handle to ProgressMonitor using the following:WBIMetadataDiscovery.getLogUtils().getProgressMonitor(). As a process elapses, you can use the setProgress() method to set progress levels that approach a specified maximum as shown below:

```
WBIMetadataConnectionImpl.getToolContext().getProgressMonitor().setMaximum(100);
WBIMetadataConnectionImpl.getToolContext().getProgressMonitor().setMinimum(0);
WBIMetadataConnectionImpl.getToolContext().getProgressMonitor().setProgress(50);
WBIMetadataConnectionImpl.getToolContext().getProgressMonitor().setNote
                              ("Getting namespace from Peoplesoft");
```

WBIMetadataDiscoveryImpl sample:

WBIMetadataDiscoveryImpl is the main entry class for invoking enterprise metadata discovery.

Interaction between the enterprise metadata discovery service and an EIS originates in an implementation of the WBIMetadataDiscoveryImpl class. You must implement the methods described below.

Constructor

The constructor should call super(Bundle Name). The bundle name is the name of the resource bundle for property-group properties of the enterprise metadata discovery; for example: super("com.abc.j2c.testapp.emd"). When creating resource bundles the EMD.properties and LogMessages.properties should share the same package name.

```
public TwineBallMetadataDiscovery() throws MetadataException {
     super("com.ibm.j2ca.sample.twineball.emd");
}
```

The WBIAdapterTypeImpl constructor requires the following parameters:

1. The name of the class representing the ResourceAdapter class.

   This is used to create property groups for ResourceAdapter in enterprise metadata discovery.
2. Number of outbound connections
3. Number of inbound connections

getAdapterTypeSummaries

Set supportedInMetadataService to true for connections that you want WebSphere Integration Developer to display when performing enterprise metadata discovery. Add the connections to adapterType using the add<Inbound/Outbound>ConnectionType() method:

```
public AdapterTypeSummary[] getAdapterTypeSummaries() throws MetadataException {
     super.getLogUtils().trace(Level.FINEST, CLASSNAME, "getAdapterTypeSummaries",
     "Enter Method");
     adapterType = new TwineBallAdapterType();
      return new WBIAdapterTypeImpl[] { adapterType };
}
```

getAdapterType

This method returns the instance of adapterType for a given ID. If the enterprise metadata discovery implementation supports only one adapterType, then it returns either that ID or the input ID.

getMetadatatree

This method returns an instance of the WBIMetadataTreeImpl implementation. Each enterprise metadata discovery implementation extends the WBIMetadataTreeImpl class and an instance of that class is returned from this method.

```
public MetadataTree getMetadataTree(MetadataConnection conn)
                              throws MetadataException {
      this.connection = (WBIMetadataConnectionImpl) conn;
      this.connType = (WBIOutboundConnectionTypeImpl)connection.getConnectionType();
      tree = new TwineBallMetadataTree(connection, super.getLogUtils());
      tree.setSelectionStyle(MetadataTree.MULTI_SELECT);
      return tree;
}
```

createServiceDescription

This class returns an instance of the inbound or outbound service description, depending on which input selection is set.

Tip: You might usefully iterate through import configurations in the MetadataSelection set and then use the properties specified on MetadataSelection to complete the service description.

The instance of ServiceDescription created should be filled in with name, namespace, function description and configurations as shown below. The method copyPropertyValues(), defined on connection configurations, copies the properties that match between the connection configuration that was used to perform

discovery and the one that used for run time. The copy is based on names. For example, if a property name Username exists in the configuration used for discovery and that used for run time, the Username value is copied.

```
public ServiceDescription createServiceDescription
        (MetadataSelection importSelection) throws MetadataException {
        ServiceDescription description = null;
        WBIMetadataSelectionImpl selection = (WBIMetadataSelectionImpl)
                               importSelection;
        MetadataImportConfiguration[] confArray = selection.getSelection();
        if (confArray.length == 0)
            return description;
        super.getLogUtils().trace(Level.FINER,
                           CLASSNAME, "createServiceDescription()",
      "Number of MetadataImportConfigurations " + confArray.length);
        PropertyGroup serviceType = selection.getAppliedSelectionProperties();
        String directionality = getDirectionality(serviceType);
        String nameSpace = getNameSpace(serviceType);
        boolean inbound = false;
        if (directionality.equals("Inbound")) {
            super.getLogUtils().trace(Level.FINER, CLASSNAME,
                               "createServiceDescription()",
          "Selected Service Type is:Inbound");
            inbound = true;
        }
        if (inbound) {
            description = createInboundServiceDescription
                   (importSelection, selection, nameSpace);
        } else {
            description = createOutboundServiceDescription
                   (importSelection, selection, nameSpace);
        }
        return description;
    }
```

setToolContext

To implement setToolContext(), initialize the LogUtils instance and set it on the Foundation Class WBIMetadataDiscoveryImpl.

```
public void setToolContext(ToolContext toolContext) {
        super.setToolContext(toolContext);
        try {
            LogUtils logUtils = new LogUtils(toolContext.getLogger(),
                       CURRENT_PACKAGE, "TwineBall Adapter", "6.0.0");
            if(logUtils != null) logUtils.trace(Level.FINER, CLASSNAME,
                       "getToolContext()", "LogUtils Initilialized");
            super.setLogUtils(logUtils);
        } catch (ResourceAdapterInternalException e) {
            e.printStackTrace();
            System.out.println(e.getMessage());
            System.out.println("Unable to create LogUtils instance");
        }
    }
```

WBIAdapterTypeImpl sample:

You use this class to implement the adapter type for enterprise metadata discovery.

Extend WBIAdapterTypeImpl and implement the methods described in the sections below.

Constructor

The constructor populates the adapter type instance as shown below:

```
public TwineBallAdapterType()throws MetadataException{
    super(Constants.RESOURCE_ADAPTER_BEAN_NAME, 2, 1);
    setId(Constants.ADAPTER_NAME);
    setDisplayName(Constants.ADAPTER_NAME);
    setDescription(WBIMetadataDiscoveryImpl.getPropertyDescription
                                       (ADAPTERTYPE_PROPERTY));
    setVendor(VENDOR);
    setVersion(VERSION);
    setOutboundConnections();
    setInboundConnections();
}
```

setInboundConnections

The `setInboundConnections()` method sets the inbound connections on the adapter type.

```
private void setInboundConnections() throws MetadataException {
    TwineBallInboundConnectionType inConnTypeForRuntime =
                new TwineBallInboundConnectionType(this);
    addInboundConnectionType(inConnTypeForRuntime);
    inConnTypeForRuntime.setResourceAdapterJavaBean
                (Constants.RESOURCE_ADAPTER_BEAN_NAME);
    inConnTypeForRuntime.setActivationSpecJavaBean
                (Constants.ACTIVATION_SPEC);
}
```

setOutboundConnections

This method sets the outbound connections on the adapter type. The connection type that can be used to perform discovery should be set to `true` for `IsSupportedInMetadataService` and connections that can be used for run time should be set to `true` for `IsSupportedAtRuntime`.

Tip: You might find it preferable to have separate connection types for enterprise metadata discovery and for run time. This way the property group for discovery can display properties needed to perform the discovery and not the entire set of properties describing `ResourceAdapter` and `ManagedConnectionFactory` properties.

```
private void setOutboundConnections() throws MetadataException {
    TwineBallOutboundConnectionType outConnTypeForRuntime =
                new TwineBallOutboundConnectionType(this);
    TwineBallOutboundConnectionType outConnTypeForMetadata =
                new TwineBallOutboundConnectionType(this);
    addOutboundConnectionType(outConnTypeForMetadata);
    addOutboundConnectionType(outConnTypeForRuntime);
    outConnTypeForMetadata.setManagedConnectionFactoryJavaBean
                (Constants.MANAGED_CONNECTION_FACTORY_NAME);
    outConnTypeForRuntime.setManagedConnectionFactoryJavaBean
                (Constants.MANAGED_CONNECTION_FACTORY_NAME);
    outConnTypeForMetadata.setResourceAdapterJavaBean
                (Constants.RESOURCE_ADAPTER_BEAN_NAME);
    outConnTypeForRuntime.setResourceAdapterJavaBean
                (Constants.RESOURCE_ADAPTER_BEAN_NAME);
    outConnTypeForMetadata.setIsSupportedInMetadataService(true);
    outConnTypeForMetadata.setIsSupportedAtRuntime(false);
    outConnTypeForRuntime.setIsSupportedInMetadataService(false);
    outConnTypeForRuntime.setIsSupportedAtRuntime(true);
}
```

WBIOutboundConnectionTypeImpl samples:

`WBIOutboundConnectionTypeImpl` represents the outbound connection types supported by the adapter. The mapping of these connection types corresponds to

the managed connection factory types that are supported by the adapter. Each managed connection factory maps to an instance of outbound connection types.

Each enterprise metadata discovery implementation should extend WBIOutboundConnectionTypeImpl and implement the methods described below.

Constructor

The constructor takes the adapterType argument, and then sets the ID, and the description and display names. If the description and display names must be globalized, they can be retrieved from the resource bundle EMD.properties using the method WBIMetadataDiscoveryImpl.getString(<prop name>).

```
public TwineBallOutboundConnectionType(WBIAdapterTypeImpl adapterType)
                                            throws MetadataException {
        super(adapterType);
        setDescription(WBIMetadataDiscoveryImpl.getPropertyDescription
                                            ("ConnectionType"));
        setDisplayName(WBIMetadataDiscoveryImpl.getPropertyName
                                            ("ConnectionType"));
        setId("TwineBall");
  }
```

createOutboundConnectionConfiguration

The createOutboundConnectionConfiguration() method returns an instance of OutboundConnectionConfiguration. Each enterprise metadata discovery implementation must extend WBIOutboundConnectionConfugurationImpl and an instance of that class should be returned in this method.

```
    public OutboundConnectionConfiguration
      createOutboundConnectionConfiguration() {
        if (conf == null) { //so we can return the same config later
            try {
                conf = new TwineBallOutboundConnectionConfiguration(this);
            } catch (MetadataException e) {
                throw new RuntimeException(e);
            }
        }
        return conf;
    }
```

WBIInboundConnectionTypeImpl samples:

WBIInboundConnectionTypeImpl represents the inbound connection types supported by the adapter. The mapping of connection types corresponds to the activationSpec types that are supported by the adapter. Each activationSpec will map to an instance of inbound connection types.

Each enterprise metadata discovery implementation should extend WBIInboundConnectionTypeImpl and implement the methods described below.

Constructor

The constructor takes the adapterType argument, setting the ID and the description and display names. You can retrieve the description and display names from the resource bundle EMD.properties using method WBIMetadataDiscoveryImpl.getString(<prop name>) (if they must be globalized).

```
public TwineBallInboundConnectionType(WBIAdapterTypeImpl adapterType)
                                            throws MetadataException {
        super(adapterType);
```

```
    setDescription(WBIMetadataDiscoveryImpl.getPropertyDescription
                                        ("ConnectionType"));
    setId("TwineBall");
    setDisplayName(WBIMetadataDiscoveryImpl.getPropertyName
                                        ("ConnectionType"));
}
```

createInboundConnectionConfiguration

The `createInboundConnectionConfiguration()` method returns an instance of `InboundConnectionConfiguration`. Each enterprise metadata discovery implementation extends `WBIInboundConnectionConfugurationImpl` and an instance of that class is returned in this method.

```
public InboundConnectionConfiguration createInboundConnectionConfiguration() {
    try {
        if (conf == null)
            conf = new TwineBallInboundConnectionConfiguration(this);
    } catch (MetadataException e) {
       throw new RuntimeException(e);
    }
    return conf;
}
```

WBIOutboundConnectionConfigurationImpl samples:

You use this class to specify outbound connection configuration properties, including those for `ResourceAdapter` and `ManangedConnectionFactory`, for your enterprise metadata discovery implementation.

Extending `WBIOutboundConnectionConfigurationImpl` requires that you implement the methods described below. Note that all methods that create instances of property groups should initialize them with the applied properties. You can do this with a utility method provided in the `EMDUtil` class as shown below after successfully constructing the property group.

```
if (getAppliedProperties() != null)
            EMDUtil.copyValues(getAppliedProperties(), adapterProp);
```

Constructor

The constructor accepts `ConnectionType`.

```
public TwineBallOutboundConnectionConfiguration
  (WBIOutboundConnectionTypeImpl connType)
    throws MetadataException {
        super(connType);
    }
```

createUnifiedProperties

The `createUnifiedProperties()` method returns an instance of property group that represents `ManagedConnectionFactory` and `ResourceAdapter` properties together in a consolidated property group. Property groups required for discovery might differ from those needed at run time. Accordingly, this method can check `isSupportedInMetadataService` and then create the corresponding property groups. WebSphere Integration Developer uses this method to display the first connection configuration screen for enterprise metadata discovery.

```
public PropertyGroup createUnifiedProperties() {
    WBIPropertyGroupImpl propGroup = null;
    try {
        propGroup =
```

```
            TwineBallConfigurationProperties.getTwineBallConfigurationProperties();
          TwineBallResourceAdapter ra =
            new TwineBallResourceAdapter();
          WBIPropertyGroupImpl adapterProp =
            (WBIPropertyGroupImpl) EMDUtil.getPropertyGroup(ra);
          propGroup.addProperty(adapterProp);
          if (getAppliedProperties() != null)
              EMDUtil.copyValues(getAppliedProperties(), propGroup);
        } catch (MetadataException e) {
          throw new RuntimeException(e);
        }
        return propGroup;
    }
```

createResourceAdapterProperties

The `createResourceAdapterProperties` method returns an instance of `PropertyGroup` that represents properties you can configure for the `ResourceAdapter` bean. The `getPropertyGroup()` method, provided in the `EMDUtil` class, fetches the properties for the base class `WBIResourceAdapter` bean. Then the enterprise metadata discovery implementation can add its own specific properties to the `ResourceAdapter` bean class.

```
public PropertyGroup createResourceAdapterProperties() {
        TwineBallResourceAdapter ra = new TwineBallResourceAdapter();
        WBIPropertyGroupImpl adapterProp = null;
        try {
            adapterProp = (WBIPropertyGroupImpl) EMDUtil.getPropertyGroup(ra);
        } catch (Exception e) {
            throw new NullPointerException(e.getMessage());
        }
        return adapterProp;
    }
```

createManagedConnectionFactoryProperties

The `createManagedConnectionFactoryProperties` method returns an instance of `PropertyGroup` that represents properties that you can configure for the `ManagedConnectionFactory` bean. The `getPropertyGroup()` method, provided in the `EMDUtil` class, fetches the properties for the base class `WBIManagedConnectionFactory` bean. As with the `ResourceAdapter` bean, the enterprise metadata discovery implementation can add its own specific properties to the `ManagedConnectionFactory` bean class.

```
public PropertyGroup createManagedConnectionFactoryProperties() {
        WBIPropertyGroupImpl connProp = null;
        try {
            connProp = ((WBIPropertyGroupImpl)
             EMDUtil.getPropertyGroup(((WBIOutboundConnectionTypeImpl)
         this.getOutboundConnectionType()).getManagedConnectionFactoryJavaBean()));
        } catch (MetadataException e) {
            throw new RuntimeException(e.getMessage());
        }
        return connProp;
    }
```

WBIInboundConnectionConfigurationImpl samples:

You use this class to specify inbound connection configuration properties, including those for `ActivationSpecWithXid`, for your enterprise metadata discovery implementation.

This class is similar to WBIOutboundConnectionConfigurationImpl except instead of `ManagedConnectionFactory`, `WBIInboundConnectionConfigurationImpl` handles the `ActivationSpecWithXid` bean class. You must extend the methods described below.

```
public PropertyGroup createActivationSpecProperties() {
        WBIPropertyGroupImpl connProp = null;
        try {
            connProp = (WBIPropertyGroupImpl) EMDUtil.getPropertyGroup
                                (new TwineBallActivationSpec());
            WBIPropertyGroupImpl credentialPropertyGroup = new WBIPropertyGroupImpl
                                            ("UserCredentials");
            credentialPropertyGroup.setDisplayName
                      (WBIMetadataDiscoveryImpl.getPropertyName("UserCredentials"));
            credentialPropertyGroup.setDescription
               (WBIMetadataDiscoveryImpl.getPropertyDescription("UserCredentials"));
            WBISingleValuedPropertyImpl property = new WBISingleValuedPropertyImpl
                                                ("UserName", String.class);
            property.setRequired(true);
            property.setDisplayName
                     (WBIMetadataDiscoveryImpl.getPropertyName("UserName"));
            property.setDescription
                     (WBIMetadataDiscoveryImpl.getPropertyDescription("UserName"));
            credentialPropertyGroup.addProperty(property);
            property = new WBISingleValuedPropertyImpl("Password", String.class);
            property.setRequired(true);
            property.setSensitive(true); //show it as **** on display
            property.setDisplayName
                     (WBIMetadataDiscoveryImpl.getPropertyName("Password"));
            property.setDescription
                     (WBIMetadataDiscoveryImpl.getPropertyDescription("Password"));
            credentialPropertyGroup.addProperty(property);
            connProp.addProperty(credentialPropertyGroup);
            property = new WBISingleValuedPropertyImpl("URL", String.class);
            property.setRequired(false);
            property.setDisplayName
                     (WBIMetadataDiscoveryImpl.getPropertyName("URL"));
            property.setDescription
                     (WBIMetadataDiscoveryImpl.getPropertyDescription("URL"));
            connProp.addProperty(property);
            property = new WBISingleValuedPropertyImpl("eventTableName", String.class);
            property.setRequired(true);
            property.setDisplayName
                     (WBIMetadataDiscoveryImpl.getPropertyName("eventTableName"));
            property.setDescription
                     (WBIMetadataDiscoveryImpl.getPropertyDescription("eventTableName"));
            property.setValue("WBIA_EVENTS");
            connProp.addProperty(property);
            connProp.addProperty(createResourceAdapterProperties());
        } catch (MetadataException e) {
            throw new RuntimeException(e);
        }
        return connProp;
    }
```

WBIMetadataTreeImpl samples:

`WBIMetadataTreeImpl` represents the object that holds the `metadataObject` nodes of the tree that WebSphere Integration Developer displays for enterprise metadata discovery.

Extend the `WBIMetadataTreeImpl` class and implement the methods described below.

Constructor

The constructor takes MetadataConnection as an argument. The constructor can also return properties from MetadataConnection that were used to start the discovery service; for example, prefix, directory name, and those properties that populate the MetadataObject nodes in the tree.

```
public TwineBallMetadataTree(WBIMetadataConnectionImpl connection,
       LogUtils logUtils)
   throws MetadataException {
        super(connection);
        this.connection = connection;
        this.logUtils = logUtils;
        try {
            cciConnection = (TwineBallConnection) connection.getEISConnection();
            twineBallConnection = cciConnection.getEISConnection();
            WBIOutboundConnectionConfigurationImpl conf =
          (WBIOutboundConnectionConfigurationImpl)
                connection.getConnectionCofiguration();
            PropertyGroup propertyGroup = conf.getAppliedProperties();
            TwineBallManagedConnection managedConnection =
          (TwineBallManagedConnection) cciConnection.getManagedConnection();
        } catch (ResourceException e) {
            throw new MetadataException(e.getMessage(), e);
        }
    }
```

createMetadataSelection

The createMetadataSelection() method returns an instance of the specific MetadataSelection class. The enterprise metadata discovery implementation extends WBIMetadataSelectionImpl and returns an instance of that class in this method.

```
    public MetadataSelection createMetaDataSelection() {
        return new TwineBallMetadataSelection();
    }
```

createFilterProperties

The createFilterProperties() method returns a property group instance that is used to perform filtering for nodes of the tree. This filter is used for displaying top level nodes on the tree only.

```
public PropertyGroup createFilterProperties() {
        WBIPropertyGroupImpl propertyGroup = null;
        try {
            propertyGroup = new WBIPropertyGroupImpl
                 (Constants.SELECTION_PROPERTIES);
            propertyGroup.setDisplayName
                 (WBIMetadataDiscoveryImpl.getPropertyName
                 (Constants.SELECTIONPROPERTIES));
            propertyGroup.setDescription
                 (WBIMetadataDiscoveryImpl.getPropertyDescription
                 (Constants.SELECTIONPROPERTIES));
            WBISingleValuedPropertyImpl typeProp = new WBISingleValuedPropertyImpl
                 (Constants.SERVICETYPE, String.class);
            String[] values = { Constants.INBOUND, Constants.OUTBOUND };
            typeProp.setValidValues(values);
            typeProp.setDefaultValue(Constants.OUTBOUND);
            typeProp.setDisplayName
                 (WBIMetadataDiscoveryImpl.getPropertyName
                 (Constants.SERVICETYPE));
            typeProp.setDescription
                 (WBIMetadataDiscoveryImpl.getPropertyDescription
```

```
            (Constants.SERVICETYPE));
        propertyGroup.addProperty(typeProp);
        WBISingleValuedPropertyImpl nameSpaceProp =
               new WBISingleValuedPropertyImpl
               (Constants.NAMESPACE, String.class);
        nameSpaceProp.setDefaultValue(Constants.TB_DEFAULT_NAMESPACE);
        propertyGroup.addProperty(nameSpaceProp);
        nameSpaceProp.setDisplayName
               (WBIMetadataDiscoveryImpl.getPropertyName
               (Constants.NAMESPACE));
        nameSpaceProp.setDescription
               (WBIMetadataDiscoveryImpl.getPropertyDescription
               (Constants.NAMESPACE));
        ServiceTypeSingleProperty operationProp =
               new ServiceTypeSingleProperty
               (Constants.OPERATIONS, String.class);
        String[] operations = TwineBallOperations.getOutboundOperations();
        operationProp.setValidValues(operations);
        operationProp.setDisplayName
               (WBIMetadataDiscoveryImpl.getPropertyName
               (Constants.OPERATIONS));
        operationProp.setDescription
               (WBIMetadataDiscoveryImpl.getPropertyDescription
               (Constants.OPERATIONS));
        propertyGroup.addProperty(operationProp);
    } catch (MetadataException e) {
        throw new RuntimeException(e);
    }
    return propertyGroup;
}
```

getMetadataObject

The `getMetadataObject()` method returns an instance of `MetadataObject` for a specific location. Each `MetadataObject` instance that is added to the `MetadataTree` should have a unique location such that when the tool calls this method, the enterprise metadata discovery implementation can find the corresponding `MetadataObject` and return it.

Tip: A sample implementation might usefully maintain a HashTable in `MetadataTree` and add the location and corresponding `MetadataObject` to it whenever a new instance of `MetadataObject` is added. Then this method could return an object corresponding to the key value from Hashtable
.

```
public MetadataObject getMetadataObject(String objectLocationID) {
    return (MetadataObject) treeNodes.get(objectLocationID);
}
```

listMetadataObjects

The `listMetadataObjects()` method returns an instance of `WBIMetadataObjectResponseImpl`. The instance should be populated with `MetadataObjects` using method `setObjects()`. The logic should use filter properties, if supported by the implementation. Any `metadataObjects` that can be selected for import should be set as `true` with the `setIsSelectableForImport()` method.

```
public MetadataObjectResponse listMetadataObjects(PropertyGroup filterParameters)
                                                       throws MetadataException {
    WBIMetadataObjectResponseImpl response = new WBIMetadataObjectResponseImpl();
```

```
    ArrayList objects = getTopLevelObjects();
    response.setObjects(objects);
    return response;
}
```

WBIMetadataObjectImpl samples:

WBIMetadataObjectImpl represents the nodes of the tree that WebSphere Integration Developer displays during enterprise metadata discovery. In most cases these nodes map to objects in the EIS that the user selects to import a service description.

WBIMetadataObjectImpl should be extended and the following methods must be implemented.

createFilterProperties

The createFilterProperties() method returns an instance of a property group that represents properties that you can use to filter searches for child objects of a MetadataObject. For example, if the top level node is a schema for JDBC, you might fetch table names with a filter consisting of a specific alphabet sequence. Implementation should return null if such properties are not applicable. For more information, see WBIMetadataTreeImpl.createFilterProperties in the Javadocs.

createObjectProperties

The createObjectProperties() method returns a property group that provides information about the specific object in MetadataTree. This helps inform the user about the metadata object instance. Implementation should return null if such properties are not applicable.

getChildren

The getChildren() method returns an instance of WBIMetadataObjectResponseImpl. The instance should be populated with child MetadataObjects using method setObjects(). (This is comparable to the MetadataTree.listMetadataObjects call.) The logic should use filter properties, if are supported by the implementation.

```
public MetadataObjectResponse getChildren(PropertyGroup filterParameters)
                                                throws MetadataException {
    WBIMetadataObjectResponseImpl response =
         new WBIMetadataObjectResponseImpl();
    ArrayList objects = getChildComponents();
    response.setObjects(objects);
    return response;
}
```

WBIMetadataSelectionImpl samples:

WBIMetadataSelectionImpl represents the object that holds the MetadataObjects that users can select for importing. This class also holds properties that users specify to select from the set of available MetadataObjects.

Extend WBIMetadataSelectionImpl and implement the following methods described below.

createSelectionProperties

The `createSelectionProperties()` method returns a property group that is used to capture inputs from users. These inputs include business object namespace, supported operations, and the relative path in the module project where XML schema definitions should be saved.

```
public PropertyGroup createSelectionProperties() {
    WBIPropertyGroupImpl propertyGroup = null;
    try {
        propertyGroup = new WBIPropertyGroupImpl
               (Constants.SELECTION_PROPERTIES);
        propertyGroup.setDisplayName
               (WBIMetadataDiscoveryImpl.getPropertyName
               (Constants.SELECTIONPROPERTIES));
        propertyGroup.setDescription
               (WBIMetadataDiscoveryImpl.getPropertyDescription
               (Constants.SELECTIONPROPERTIES));
        WBISingleValuedPropertyImpl typeProp =
                createServiceTypeProperty(propertyGroup);
        ServiceTypeSingleProperty operationProp =
                createNamespaceProperty(propertyGroup);
  createMaxRecordProperty(propertyGroup);
  createRelativePathProperty(propertyGroup);
        //Copy the applied properties to the new instance
        if (this.getAppliedSelectionProperties() != null)
            EMDUtil.copyValues
            (this.getAppliedSelectionProperties(), propertyGroup);
        typeProp.addPropertyChangeListener(operationProp);
    } catch (MetadataException e) {
        throw new RuntimeException(e);
    }
    return propertyGroup;
}
```

WBIMetadataImportConfigurationImpl samples:

`WBIMetadataImportConfigurationImpl` represents an instance of `MetadataObject` and the configuration properties that users specify for it.

Extend `WBIMetadataImportConfigurationImpl` and implement the methods described below.

Constructor

The constructor for `WBIMetadataImportConfigurationImpl` accepts `MetadataObject`.

```
public TwineBallMetadataImportConfiguration(WBIMetadataObjectImpl
metadataObject) { super(metadataObject); }
```

createConfigurationProperties

The `createConfigurationProperties()` method returns a property group for the `MetadataObject`. The properties are specific to the instance of `MetadataObject` and are used to capture inputs from users. Those inputs might include operations that are supported for each `MetadataObject` instance, or additional information required to process the object at run time.

WBIMetadataEditImpl samples:

The enterprise metadata discovery service uses WBIMetadataEditImpl to acquire connectionTypes, which contains editable properties forResourceAdapter, ManagedConnectionFactory, or ActivationSpecWithXid.

The enterprise metadata discovery tooling creates an instance of WBIMetadataEditImpl during the boot strap process. Along with the name of the MetadataDiscovery class, WBIMetadataEditImpl is specified in the discovery-service.xml file. If for any reason WBIMetadataEditImpl cannot be instantiated, then enterprise metadata discovery is not selectable in the WebSphere Integration Developer tooling.

getOutboundConnectionType

The getOutboundConnectionType() method returns an instance of OutboundConnectionType with an input name.

```
public OutboundConnectionType getOutboundConnectionType(String arg0)
                                          throws MetadataException {
    return adapterType.getOutboundConnectionTypes()[1];
}
```

getInboundConnectionType

The getInboundConnectionType() method returns an instance of InboundConnectionType with an input name.

```
public InboundConnectionType getInboundConnectionType(String arg0)
                                        throws MetadataException {
    return adapterType.getInboundConnectionTypes()[0];
}
```

WBIDataDescriptionImpl samples:

WBIDataDescriptionImpl represents the data description interface. This interface maps business object definitions to Java™ objects.

prepareChildSchemaFiles

The prepareChildSchemaFiles() method creates child objects when a parent object cannot do so. If your adapter requires a separate and recursive creating of child objects, then implement this method. Its should call recursively as shown:

```
prepareChildSchemaFiles();
prepareSchemaFiles();
```

```
public void prepareChildSchemaFiles() throws MetadataException {
    WBIMetadataDiscoveryImpl.getLogUtils().trace(Level.FINER, CLASSNAME,
                                      "prepareChildSchemaFiles",
       "Entering Method");
    MetadataObjectResponse response = getMetadataObject().getChildren(null);
    WBIMetadataDiscoveryImpl.getLogUtils().trace(Level.FINER,
                                CLASSNAME, "prepareChildSchemaFiles",
 "Number of BUSINESS OBJECTS: " + response.getObjectIterator().size());
    for (Iterator i = response.getObjectIterator(); i.hasNext();) {
        TwineBallMetadataObject bo = (TwineBallMetadataObject) i.next();
        TwineBallDataDescription dataDesc = new TwineBallDataDescription();
        dataDesc.setMetadataObject(bo);
 dataDesc.setRelativePath(getRelativePath());
        dataDesc.setName(getName().getNamespaceURI(), bo.getBOName());
        WBIMetadataDiscoveryImpl.getLogUtils().trace(Level.FINER,
                                             CLASSNAME,
    "prepareChildSchemaFiles",
    "Build Child Business Object : " + bo.getBOName());
        if (bo.getType() == WBIMetadataObjectImpl.OBJECT)
```

```
            dataDesc.prepareChildSchemaFiles();
        WBIMetadataDiscoveryImpl.getLogUtils().trace(Level.FINER,
                                                      CLASSNAME,
    "preparingChildSchemaFiles",
    "Preparing SchemaFile for " + bo.getDisplayName());
        dataDesc.prepareSchemaFiles();
        SchemaDefinition[] schemaFiles = dataDesc.getSchemaDefinitions();
        for (int j = 0; j < schemaFiles.length; j++) {
            SchemaDefinition definition = schemaFiles[j];
            put(definition.getNamespace(), definition.getLocation(),
                                           definition.getContent());
        }
    }
    WBIMetadataDiscoveryImpl.getLogUtils().trace(Level.FINER, CLASSNAME,
                                           "prepareChildSchemaFiles",
       "Exiting Method");
}
```

getVerbs

The `getVerbs()` method returns a list of verbs when the data description represents a top-level object that supports verbs.

```
public List getVerbs() {
     ArrayList list = new ArrayList();
     list.add(TopLevelVerbs.CREATE_TLV);
     list.add(TopLevelVerbs.UPDATE_TLV);
     list.add(TopLevelVerbs.DELETE_TLV);
      return list;
}
```

getMetadataForAttribute

The `getMetadataForAttribute()` method returns an instance of the `WBIMetadata` object and represents application specific information for the element or field in the object definition. This corresponds to the annotation section for the element definition.

```
public WBIMetadata getMetadataForAttribute(String attrName) {
    WBIMetadata attributeMetadata = new WBIMetadata();
    attributeMetadata.setSource(Constants.ASI_TARGET_NAMESPACE);
    attributeMetadata.setObjectNameSpace(Constants.ATTR_APPINFO_ASI_TYPE_TAG);
    QName asiNamespace = new QName
             (Constants.ASI_TARGET_NAMESPACE, Constants.ASI);
    attributeMetadata.setNameSpace(asiNamespace);
    attributeMetadata.setASI(Constants.FIELD_NAME, attrName);
    if (attrName.equalsIgnoreCase(Constants.PRIMARYKEY)) {
        attributeMetadata.setASI(Constants.PRIMARY_KEY, "true");
    }
    return attributeMetadata;
}
```

getMetadataForBusinessObject

The `getMetadataForBusinessObject()` method returns an instance of the `WBIMetadata` object and represents application specific information for the object definition. This corresponds to the annotation section for the `complexType` definition.

```
public WBIMetadata getMetadataForBusinessObject() {
    WBIMetadataDiscoveryImpl.getLogUtils().traceMethodEntrance
                  (CLASSNAME, "getMetadataForBusinessObject");
    WBIMetadata bometadata = new WBIMetadata();
    bometadata.setSource(Constants.ASI_TARGET_NAMESPACE);
    QName namespace = new QName(Constants.ASI_TARGET_NAMESPACE, Constants.ASI);
```

```
        bometadata.setNameSpace(namespace);
        bometadata.setObjectNameSpace(Constants.BUS_OBJ_APPINFO_ASI_TYPE_TAG);
        bometadata.setASI(Constants.ASI_OBJECTNAME,
                       this.getMetadataObject().getDisplayName());
        WBIMetadataDiscoveryImpl.getLogUtils().traceMethodExit
                       (CLASSNAME, "getMetadataForBusinessObject");
        return bometadata;
    } // End of BO level metadata
```

isContainer

The `isContainer()` method returns `true` when the complexType definition requires a Container definition with the graph definition. The Container definition is used when the adapter supports a `RetrieveAll` operation.

```
public boolean isContainer() {
        WBIMetadataDiscoveryImpl.getLogUtils().traceMethodEntrance
                                       (CLASSNAME, "isContainer");
        boolean retValue = false;
        String objName = getName().getLocalPart();
        if (objName.endsWith("Container") && !objName.equals(getBOName()))
            retValue = true;
        WBIMetadataDiscoveryImpl.getLogUtils().traceMethodExit
                                      (CLASSNAME, "isContainer");
        return retValue;
    }
```

getType

The `getType()` method returns the XML schema definition type that represents the element or field of the object definition.

```
    public String getType(String attrName) {
        if (getCardinality(attrName) == Constants.N_CARDINALITY) {
            return attrName.toLowerCase() + ":"
               + StringCaseChanger.toCamelCase(attrName);
        }
        return Constants.XS_STRING;
    }
```

getAttributeName

The `getAttributeName()` method returns the element name that represents the field in the object. This is the element name that would be used in the XML schema definition.

```
    public String getAttributeName(String attrName) {
        return StringCaseChanger.toCamelCase(attrName);
    }
```

getImportNameSpaces

The `getImportNameSpaces()` method returns a list of `ImportedNameSpace` instances that should be imported in the business object schema.

```
    public List getImportNameSpaces() throws MetadataException {
        WBIMetadataDiscoveryImpl.getLogUtils().traceMethodEntrance
                               (CLASSNAME, "getImportNameSpaces");
        ArrayList list = new ArrayList();
        ImportedNameSpace namespace = new ImportedNameSpace();
        namespace.setLocation(Constants.TWINEBALLASI_XSD);
        namespace.setNameSpace(Constants.ASI_TARGET_NAMESPACE);
        list.add(namespace);
```

```
        WBIMetadataDiscoveryImpl.getLogUtils().traceMethodExit
                                (CLASSNAME, "getImportNameSpaces");
        return list;
    }
```

getNameSpaces

The getNameSpaces() method returns the NameSpaces listed in the XML schema definition. Typically these are application specific information schema definition namespaces. Note that this list is for outside namespaces only; child object namespaces are included by the Adapter Foundation Classes.

```
    public List getNameSpaces() {
        WBIMetadataDiscoveryImpl.getLogUtils().traceMethodEntrance
                                      (CLASSNAME, "getNameSpaces");
        ArrayList list = new ArrayList();
        list.add(new QName(Constants.ASI_TARGET_NAMESPACE, Constants.ASI));
        WBIMetadataDiscoveryImpl.getLogUtils().traceMethodExit
                                      (CLASSNAME, "getNameSpaces");
        return list;
    }
```

getASISchemaName

The getASISchemaName() method returns the target namespace that represents the application specific information schema.

```
    public String getASISchemaName() {
        return Constants.ASI_TARGET_NAMESPACE;
    }
```

getCardinality

The getCardinality() method returns the cardinality for an elements in the business object definition. This value is used to formulate the maxOccurs and minOccurs tag.

```
    public String getCardinality(String attrName) {
        TwineBallAttribute definition =
          (TwineBallAttribute) this.getAttributeList().get(attrName);
        if (definition.isChild)
            return Constants.N_CARDINALITY;
        else
            return "1";
    }
```

getMaxLength

The getMaxLength() method returns the maxLength for the elements in the business object definition. This is used to fill in the maxLength tag for the XML schema definition.

```
    public int getMaxLength(String name) {
        return 0;
    }
```

getRequired

The getRequired() method returns true if the element is marked as required in the XML schema definition. Otherwise, this method returns false.

```
public boolean getRequired(String attrName) { return false; }
```

getChildList

The getChildList() method returns the Iterator for the child objects of the DataDescription.

```
public Iterator getChildList() throws MetadataException
{
   return (this.getMetadataObject().getChildren(null)).getObjectIterator();
}
```

WBIInboundServiceDescriptionImpl samples:

WBIInboundServiceDescriptionImpl represents the object that populates function descriptions for inbound service descriptions.

Implement the method shown in the section below.

setFunctionDescriptions

The setFunctionDescriptions() method populates function descriptions based on objects and properties selected in MetadataSelection .

```
public void setFunctionDescriptions(MetadataSelection selection)
                                       throws MetadataException {
    MetadataImportConfiguration[] selectedObjects = selection.getSelection();
    PropertyGroup selectionProperties =
        ((WBIMetadataSelectionImpl) selection).getAppliedSelectionProperties();
    WBIMultiValuedPropertyImpl operationsProperty =
         (WBIMultiValuedPropertyImpl) selectionProperties.getProperty("Operations");
    String[] operations = operationsProperty.getValuesAsStrings();
    ArrayList functionDescriptions = new ArrayList();
    String location = TwineBallConfigurationProperties.getLocation
                                            (selectionProperties);

    // iterate through the objects.
    for (int i = 0; i < selectedObjects.length; i++) {
        WBIMetadataImportConfigurationImpl spec =
               (WBIMetadataImportConfigurationImpl) selectedObjects[i];
        WBIInboundFunctionDescriptionImpl functionDescription;
        TwineBallMetadataObject metadataObj =
            (TwineBallMetadataObject) spec.getMetadataObject();
        for (int j = 0; j < operations.length; j++) {
            String operation = operations[j];
            functionDescription = new WBIInboundFunctionDescriptionImpl();
            char firstCharacter = operation.charAt(0);
            //convention for displaying the outbound function.
            functionDescription.setName
                  ("emit" + Character.toUpperCase(firstCharacter) +
                      operation.substring(1).toLowerCase() + "AfterImage" +
 StringCaseChanger.toCamelCase(metadataObj.getDisplayName()));
            functionDescription.setEISFunctionName(functionDescription.getName());
            functionDescription.setImportConfiguration(spec);
            TwineBallDataDescription dataDescription = new TwineBallDataDescription();
            dataDescription.setMetadataObject(metadataObj);
            dataDescription.setName(getNameSpace(),
              StringCaseChanger.toCamelCase(metadataObj.getDisplayName()));
            dataDescription.populateSchemaDefinitions();
            dataDescription.setRelativePath(location);
            dataDescription.setName(getNameSpace() + "/" +
                      metadataObj.getBOName().toLowerCase() +
                         "bg", metadataObj.getBOName() + "BG");
            functionDescription.setInputDataDescription(dataDescription);
            functionDescriptions.add(functionDescription);
        }
    }
```

```
        FunctionDescription[] funcArray =
            new FunctionDescription[functionDescriptions.size()];
        functionDescriptions.toArray(funcArray);
        super.setFunctionDescriptions(funcArray);
    }
```

WBIOutboundServiceDescriptionImpl samples:

`WBIOutboundServiceDescriptionImpl` represents the object that populates function descriptions for outbound service descriptions.

Implement the method shown in the section below.

setFunctionDescriptions

The `setFunctionDescriptions()` method populates the function descriptions based on objects and properties selected in `MetadataSelection`.

```
public void setFunctionDescriptions(MetadataSelection selection)
                            throws MetadataException {
    WBIMetadataDiscoveryImpl.getLogUtils().traceMethodEntrance
                            (CLASSNAME,SETFUNCTIONDESCRIPTIONS);
    ArrayList functionDescriptions = new ArrayList();
    MetadataImportConfiguration[] supportedObjects = selection.getSelection();
    PropertyGroup selectionProperties =
           ((WBIMetadataSelectionImpl) selection).getAppliedSelectionProperties();
    WBIMultiValuedPropertyImpl operationProperty =
            (WBIMultiValuedPropertyImpl) selectionProperties.getProperty("Operations");
    String[] supportedOperations = operationProperty.getValuesAsStrings();
    traceFiner("supportedOperations=" + supportedOperations);
    String location = TwineBallConfigurationProperties.getLocation(selectionProperties);

    for (int i = 0; i < supportedObjects.length; i++) {
 WBIMetadataImportConfigurationImpl importConfiguration =
            (WBIMetadataImportConfigurationImpl) supportedObjects[i];
 WBIOutboundFunctionDescriptionImpl outboundFunctionDescription;
 WBIInteractionSpec interactionSpec;
 TwineBallMetadataObject metadataObj = (TwineBallMetadataObject)
                            importConfiguration.getMetadataObject();
 traceFiner("Object name is " + metadataObj.getBOName());
 for (int j = 0; j < supportedOperations.length; j++) {
     String operation = (String) supportedOperations[j];
     traceFiner("generating function for the " + operation
             + " operation on " + metadataObj.getBOName());
     outboundFunctionDescription = new WBIOutboundFunctionDescriptionImpl();
     outboundFunctionDescription.setName(operation.toLowerCase()
                                 + metadataObj.getDisplayName());
     TwineBallDataDescription dataDescription = new TwineBallDataDescription();
     dataDescription.setMetadataObject(metadataObj);
     dataDescription.setName(getNameSpace(), metadataObj.getDisplayName());
     dataDescription.populateSchemaDefinitions();
     dataDescription.setRelativePath(location);
     outboundFunctionDescription.setInputDataDescription(dataDescription);
     dataDescription.setName(getNameSpace() +
                   "/" + metadataObj.getBOName().toLowerCase() +
      "bg", metadataObj.getBOName() + "BG");
     traceFiner("type of input object" +
            dataDescription.getName().getNamespaceURI() +
     "\\" + dataDescription.getName().getLocalPart());
     if (operation != WBIInteractionSpec.RETRIEVE_ALL_OP)
  outboundFunctionDescription.setOutputDataDescription(dataDescription);
     else {
  dataDescription = new TwineBallDataDescription();
  dataDescription.setName(getNameSpace(),
          metadataObj.getDisplayName() + "Container");
```

```
  dataDescription.setMetadataObject(metadataObj);
  dataDescription.populateSchemaDefinitions();
  dataDescription.setRelativePath(location);
  dataDescription.setName(getNameSpace() +
      "/" +
      metadataObj.getBOName().toLowerCase() +
      "container", metadataObj.getBOName() +
      "Container");
  outboundFunctionDescription.setOutputDataDescription(dataDescription);
     }
     interactionSpec = new WBIInteractionSpec();
    if (operation.equals(WBIInteractionSpec.RETRIEVE_ALL_OP)) {
     MaxRecordSingleProperty maxCount =
            (MaxRecordSingleProperty) selectionProperties.getProperty
                             (TwineBallMetadataSelection.MAXRECORDS);
    if (maxCount.getValue() == null)
 interactionSpec.setMaxRecords(((Integer)
            maxCount.getPropertyType().getDefaultValue()).intValue());
    else
 interactionSpec.setMaxRecords(((Integer) maxCount.getValue()).intValue());
    }
  interactionSpec.setFunctionName(operation);
     outboundFunctionDescription.setInteractionSpec(interactionSpec);
     outboundFunctionDescription.setImportConfiguration(importConfiguration);
     functionDescriptions.add(outboundFunctionDescription);
 }
    }
    FunctionDescription[] functionArray =
                         new FunctionDescription[functionDescriptions.size()];
    functionDescriptions.toArray(functionArray);
    super.setFunctionDescriptions(functionArray);
    WBIMetadataDiscoveryImpl.getLogUtils().traceMethodEntrance
                        (CLASSNAME,SETFUNCTIONDESCRIPTIONS);
}
```

Enterprise Metadata Discovery interfaces and implementation for technology adapters

Version 1.1 of Enterprise Metadata Discovery includes enhancements for configurable data handlers, function selectors, and data bindings, and a way to *build* service descriptions using these configured artifacts and existing schemas.

The enhancements to Enterprise Metadata Discovery are useful when building technology-style adapters that transform unstructured data to structured data and make use of existing schemas instead of generating them from an EIS system.

Enterprise Metadata Discovery and technology-style adapters use data handlers, function selectors and data bindings in the following ways:

- A data handler, which is a Java class or library of classes, is used by a process to transform data into and from specific formats. In the WebSphere business integration environment, data handlers transform text data of specified formats into business objects, and transform business objects into text data of specified formats.
- A function selector is used by adapters performing inbound processing. The function selector directs the data to the appropriate function in the message-driven bean (MDB) or EIS export. A function selector may inspect data or metadata to determine how and where to direct the data.
- A data binding, which is a Java class, converts a stream of native data to a business object during inbound processing, and converts a business object to a stream of native data during outbound processing.

Creating services that use technology-style adapters relies on being able to implement the interfaces in the `commonj.connector.metadata.build.*` package or by extending the adapter foundation classes in `com.ibm.j2ca.extension.emd.build.*` package, or a combination thereof.

Building configurable artifacts (data bindings, data handlers, and function selectors)

Building services using Enterprise Metadata Discovery involves building configurable artifacts that include data bindings, data handlers and function selectors.

Implementing a data handler

To implement a data handler, implement the `common.connector.runtime.DataHandler` interface.

This interface contains two methods as follows:

- `Object transform(Object input, Class targetClass, Object options)`
- `void transformInto(Object input, Object target, Object options)`

The `transform` method takes an InputStream and produces a data object from it or takes a data object and produces an InputStream (depending on the processing mode). The target class indicates the processing to be performed by the data handler. The options parameter is a map that can include an *encoding* parameter. When implementing a data handler, code it to handle both byte array and string formats, if possible.

Note: Existing Websphere adapters, as of version 6.1.x and version 6.2, use InputStream for accessing raw data and data object for the transformed version exclusively.

Implement the `TransformInto` method in a data handler to transform a data object by writing to an output stream, so the input would be a data object, and the target would be an output stream.

Note: A data handler should only transform from the raw *payload* to a data object and vice-versa; it should not put the data in a record, or an envelope or other adapter-specific format, as this is the processing responsibility of a data binding.

Implementing a function selector

When provided data and metadata from and adapter, the function selector generates the native function name. A function selector implements the interface `commonj.connector.runtime.FunctionSelector`. This interface contains the method String `generateEISFunctionName(Object[] args);`.

The argument array you get depends on which listener interface you are using. If you are using `InboundListener` (passing data only), the first object in the array is the record. If you are using `ExtendedMessageListener` (passing data and metadata), the first object in the array is an `InboundNativeDataRecord` that contains an `InboundInteraction` spec (for metadata) and the record (for data).

For services using adapters to exchange data with a local file system (like flat files) and services using adapters to access files on an FTP server, the `InboundInteractionSpec` contains the filename and path to the files being

processed. A custom function selector can use the information in the InboundInteractionSpec to generate a native function.

For a custom adapter, you can create either a *smart* function selector that utilizes the data or metadata to perform function selection, with or without configuration; or you can implement a *simple* function selector that always returns a static function name. Implementing a simple function selector restricts your inbound service to a single function. This, in turn, restricts the service to dealing with a single data type; unless *anyType* is used in the WSDL, which is discouraged because a WSDL that contains an *anyType* schema is difficult to use in WebSphere Process Server or WebSphere ESB (due to the fact that it does not indicate the actual type emitted).

Implementing a data binding

A data binding implements the interface commonj.connector.runtime.RecordHolderDataBinding. This interface contains four methods as follows:

- `Record getRecord();`
- `void setRecord(Record record);`
- `DataObject getDataObject();`
- `void setDataObject(DataObject object);`

These methods perform the mediation between the record that the adapter needs (for example an InputStreamRecord), and the data object specified in the service (WSDL).

The data binding can call a data handler to perform low-level transformation on an input stream. The adapter foundation classes provide a base class, named BaseDataBinding that adapters can extend to handle the low-level details of calling a data handler. The BaseDataBinding data handler has three methods as follows:

- `InputStream transformToInputStream(DataObject object);`
- `DataObject transformToDataObject(InputStream stream, QName expectedType);`
- `void setChildDataHandler(Qname bindingConfiguration);`

To use these methods, call setChildDataHandler first with a data handler configuration, then call one of the transform methods. The **expectedType** parameter on transformToDataObject should match the type of the DataObject you want the transformation to produce. You can derive this from the expected type that the data binding is passed in the BindingContext interface described in the Binding context and configuration.

Binding context and configuration

Data handlers, data bindings and function selectors can be context-enabled and may be configurable.

BindingContext is a *mix-in* interface that provides access to contextual information, such as the configuration of this binding, the expected type for data handlers and data bindings and the type of service being used.

The BindingContext interface has several constants and one method as follows:

- `void setBindingContext(Map context);`

To access the binding configuration, do `context.get(BindingContext.BINDING_CONFIGURATION)`

To access the expected type, do `context.get (BindingContext.EXPECTED_TYPE)`

Function selectors, data handlers, and data bindings are all configurable. This configuration can be quite rich, providing single and multi-valued properties, drop-downs, and user-editable tables and trees.

Every configurable artifact needs to have a **Properties** bean. The class name for this bean must follow the naming convention `[Artifact]Properties`, where `[Artifact]` is the class name of the function selector, data binding, or data handler being configured. The properties bean contains all the artifact's properties as JavaBean properties. Getters and setters must be provided for every attribute. Arrays are allowed for complex properties.

To enable the type of configuration that provides single and multi-valued properties, drop-downs, and user-editable tables and trees, you need a `Configuration` class and an EditableType class; each one using the same naming convention as the Properties class: `[Artifact]Configuration` and `[Artifact]EditableType`, respectively. The `Configuration` class must implement the interface `commonj.connector.metadata.BindingConfigurationEdit`, which has one method of note, which is `public EditableType getEditableType();`.

The method `public EditableType getEditableType();` return the EditableType implementation for this artifact. The EditableType class implements the `commonj.connector.metadata.discovery.EditableType` interface, which contains the following methods:

- `PropertyGroup createProperties()`
- `void synchronizeFromBeanToPropertyGroup(Object bean, PropertyGroup pg)`
- `void synchronizeFromPropertyGroupToBean(PropertyGroup pg, Object bean)`

The `createProperties` method creates an enterprise metadata discovery (EMD) property group that can contain several properties, or even nested property groups.

The `synchronization` methods provide the capability to keep the property group synchronized with the property bean. To specify a data handler property in a data binding, use a `BindingTypeBeanProperty` in the Property bean and a `WBIBindingProperty` in the property group.

Implementing Enterprise Metadata Discovery to build an interface

The `com.ibm.j2ca.extension.emd.build.*` package allows a simpler way to build services with an adapter from existing types. Instead of the EMD process generating types, it will import them and use them in a service.

Extending the adapter foundation classes to build services

Creating services that use technology-style adapters relies on being able to implement the interfaces in the commonj.connector.metadata.build.* package or by extending the AFC classes in com.ibm.j2ca.extension.emd.build.* package, or a combination thereof.

To implement the interfaces that allow you to build services, you need to extend the following adapter foundation classes:

- `WBIMetadataBuild`
- `WBIFunctionBuilder`
- `WBIMetadataType` (optional)

When you extend `WBIMetadataBuild`, you will need to implement the following methods:

- `FunctionBuilder createFunctionBuilder(String functionSelector)`

  `CreateFunctionBuilder` returns your FunctionBuilder instance.
- `String[] getConnectionSpecClassName();`

  `getConnectionSpecClassName` returns the class name of your J2CA connection spec.
- `InteractionSpec getDefaultInteractionSpec();`

  `getDefaultInteractionSpec` returns the class name of the J2CA interaction spec that is most commonly used for your adapter.
- `QName[] getAdapterSchemaTypes();`

  `getAdapterSchemaTypes` returns any common schemas; for example, schemas that are used independently of the operation.
- `SchemaDefinition[] getSchemasForQName(QName type);`

  `getSchemasForQName` returns the SchemaDefinition[s] for any Qnames that you provide for schemas in the build process.

When you extend `WBIFunctionBuilder`, you will need to implement the following methods:

- `FunctionType[] getFunctionTypes();`

  This method returns the function type(s) for the selected operation. This is particularly useful for determining whether this is input/output, the input or output are the same or different.
- `String getDefaultDataBindingClassName();`

  This method returns the default data binding class for this adapter.
- `updateInputDataDescription(DataDescription dataDescription, FunctionDescription functionDescription);`

  This method gives your adapter a chance to update the data description for input.
- `String[] getSupportedOperationNames();`

  This method returns the list of operations that the adapter foundation classes display to the user. The user can select one of these per added function.
- `String[] getRecordInterfaces();`

  This method returns the record interfaces that your adapter can deal with, for instance InputStreamRecord.
- `InteractionSpec getInteractionSpec(String methodName, QName inputData, QName outputData)`

  This method returns a populated the interaction spec, containing the necessary information about the function to be run. This is used in building the service description.

WBIMetadataType

If your adapter requires the ability to generate its own types (or wrappers), you should either extend `WBIMetadataType` or implement the `MetadataType` interface directly.

Extend `WBIMetadataType` if your adapter needs a simple *wrapper* object around a payload object, similar to the IBM WebSphere Adapter for Flat Files and the IBM WebSphere Adapter for FTP. The `WBIMetadataType` interface allows you to select a payload type, and optionally generate a business graph structure in addition to a plain wrapper. Implement `MetadataType` interface directly if you need to deal with a more complex object structure.

To extend `WBIMetadataType`, implement the following methods:

- `public abstract String getDefaultNamespace();`

  This method returns a default namespace for your adapter.
- `public abstract SchemaDefinition[] getSchemaDefinitions();`

  This method returns the generated schema definitions, based on the payload. The following helper methods enable you to do this:
  - `getImportedSchemaLocationString();`

    This helper method returns the relative location of the selected schema.
  - `getImportedSchema()`

    This helper method returns the **Qname** of the selected schema.
  - `getNamespace()`

    This helper method returns the namespace that the user has entered.

Discovery-service.xml

For the tool to detect and use your adapter, you need a discovery-service.xml file in yor “meta-in” folder.

Here is an example of the discovery-service.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
 <emd:discoveryService xmlns:emd="commonj.connector"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/j2ee_1_4.xsd">
<j2ee:description>My Adapter</j2ee:description>
<j2ee:display-name>My Adapter</j2ee:display-name>

  <emd:vendor-name xsi:type="j2ee:xsdStringType">My vendor </emd:vendor-name>
  <emd:version xsi:type="j2ee:xsdStringType">My Version</emd:version>
  <emd:spec-version>1.1</emd:spec-version>
  <emd:metadataBuild-class xsi:type="j2ee:fully-qualified-classType">
   mypackage.myadapter.MetadataBuild</emd:metadataBuild-class>
  <emd:metadataEdit-class xsi:type="j2ee:fully-qualified-classType">
   mypackage.myadapter.MetadataEdit</emd:metadataEdit-class>
  <emd:metadataType-class xsi:type="j2ee:fully-qualified-classType">
   mypackage.myadapter.MetadataType</emd:metadataType-class>
 <emd:application-specific-schema>
  </emd:application-specific-schema></emd:discoveryService>
```

Structured record implementation

StructuredRecord class needs to be implemented by adapters when the data exchanged with backend application can be well defined. Extend foundation class `com.ibm.j2ca.base.WBIStructuredRecord` and implement the associated methods.

Initialize input method

This method resolves the type of the metadata if it's a JavaBean or SDO type and initializes the metadata interfaces appropriately. `public void initializeInput(DataExchangeFactory dataBinding, Object metadata) throws DESPIException` .

Purpose of the initialize input method

Implement the `initializeInput` method only if the metadata contains information to be used to initialize the back-end connection for processing the request. For example, if an application requires you to instantiate a corresponding component on the back-end application to process the request.

This method should first invoke `super.initializeInput()` to initialize the cursors, accessors and metadata.

Sample code

Here is a coding sample on how to implement the `intitializeInput` method:

```
public void initializeInput(DataExchangeFactory factory, Object[] metadata) throws DESPIException {
  super.initializeInput(factory, metadata);
  objectNaming = new ObjectNaming();
  objectSerializer = new ObjectSerializer(objectNaming);
  try{
  objectAnnotations = super.getMetadata()
.getAnnotations(
TwineBallConstants.METADATA_NAMESPACE);
  } catch(Exception e){
   throw new DESPIException(e);
  }
 }
```

Initialize output method

`public void initializeOutput(DataExchangeFactory dataBinding, Object metadata) throws DESPIException`.

Purpose of the initialize output method

Implement this method if there is some initialization needed to support record retrieval from `getNext()`.

This method should first invoke `super.initializeInput()` to initialize the cursors, accessors and metadata.

Sample code

Here is a coding sample on how to implement the `intitializeOutput` method:

```
public void initializeOutput(DataExchangeFactory factory, Object[] metadata) throws DESPIException {
  super.initializeOutput(factory, metadata);
  objectNaming = new ObjectNaming();
  objectSerializer = new ObjectSerializer(objectNaming);
  try{
   objectAnnotations = super.getMetadata().getAnnotations(TwineBallConstants.METADATA_NAMESPACE);
  }catch(Exception e){
   throw new DESPIException(e);
  }

 }
```

Set managed connection method

This method passes the ManagedConnection handle to the record implementation, allowing the record to get access to the physical connection to the backend application to perform processing.public void setManagedConnection(ManagedConnection managedConnection) throws ResourceException

Purpose of the set managed connection method

This method is called after initializeInput().

Implement this only if there is a need for to build a component on the backend EIS to perform an adapter function. For example, if the backend application needs to initialize a corresponding component on the backend EIS, use this method to initialize that component.

Sample code

Here is a coding sample on how to implement the setManagedConnection method:

```
public void setManagedConnection(
   ManagedConnection managedConnection) throws ResourceException {

  try {
   this.managedConnection = managedConnection;
   if (this.getEISRepresentation() == null) {
    //get the handle to physical connection
   BackEndConnection conn =
 managedConnection.getBackEndHandle();
    Object object = conn.create(name);
    if (object == null) {
     throw new ResourceException("Invalid metadata defined for the input Data.");
  }
  //set the instantiated object, would be accessed later
    this.setEISRepresentation(object);
   }
  } catch (Exception e) {
   throw new ResourceException("Failed in creating object " + name);
  }
 }
```

Get next method

The client application invokes this method to retrieve data through the adapter. For outbound processing, this method is used to *get* the response data back from the adapter. For inbound processing, this method is called to *get* the inbound data from the adapter.public boolean getNext(boolean copyValues) throws DESPIException

Purpose of the getNext method

Use the argument copyValues to fill in values as part of getNext.

When the value for copyValues is set to True, the getNext() method should fill in data in output cursors as part of the call.

When the value for copyValues is set to False, the getNext() method should just keep the instance of cursors ready, but not fill in the data. The data would be filled in using the pushValue() call, which passes an Xpath expression.

This method can be called multiple times by the client application if the operation can return multiple records from the backend application. For example, if

the retriveAll operation could return "N" records from the backend application, for each call to the getNext() method the implementation should fill in data of one record from the backend application into OutputCursors/Accessors. It should keep track of which record it needs to do next so in subsequent call to getNext() it can fill in next record.

This method would be using the eisRepresentation being held on to the Record instance, this is the backend representation of data which is used in this method to read fields and set those in Output Cursors and Accessors.

Sample code

For a sample of how to implement the `getNext()` method, refer to the TwineBall sample.

Clone method

Use this method to copy property values from the record instance to a newly created instance.`public Object clone ()`

Sample

Here is a coding sample on how to implement the `Clone` method:

```
public Object clone()  {
//Build a new record
TwineBallStructuredRecord record = new TwineBallStructuredRecord();
try{
//Copying property values
 record.twineBallConnection = twineBallConnection;
 record.objectNaming = objectNaming;
 record.objectSerializer = objectSerializer;
 record.setEISRepresentation(this.getEISRepresentation());
 }catch(Exception e){
  throw new RuntimeException(e);
 }
 return record;
}
```

Close method

This method provides cleanup for the `getNext method.public void close()`.

Purpose of the close method

Implement this method if you need to release resources/objects after the `getNext()` method has been run.

Sample code

Here is a coding sample showing how to implement the `Close` method:

Extract method

`public void extract(String xpath) throws DESPIException` What is this method? Need a short description.

Purpose

This method needs implementation only if an Xpath expression can be used to retrieve individual field values from backend object representation. This method is invoked after getNext() is called with boolean value as false.

The method should first extract the value from backend object representation defined through Xpath and use OutputCursor and OutputAccessor interfaces to populate values in runtime data structure.

Sample

Data binding implementation

Adapters must provide implementations for DataBinding interface in order to work with WebSphere Process Server. The marshalling of data from SDO to CCI record and from CCI record to SDO occurs through DataBinding implementation.

Interfaces

Adapters should implement the following interfaces:

- commonj.connector.runtime
- RecordHolderDataBinding

The following sections describe the methods that need implementation.

setDataObject

```
public void setDataObject(DataObject arg0) throws DataBindingException
```

This method builds an instance of adapter record instance and initializes with the metadata that represents input SDO.

```
public void setDataObject(DataObject arg0) throws DataBindingException
{
 // TODO Auto-generated method stub
 try {
  record = new TwineBallStructuredRecord();
  inputBG = arg0;
  DEFactorySDO binding = new DEFactorySDO();
  DataObject dataObject = arg0.getDataObject(WPSServiceHelper.getRootBusinessObjectProperty
 (arg0.getType()));
  binding.setBoundObject(dataObject);
  record.initializeInput(binding, dataObject);
 }
 catch (Exception e) {
  throw new DataBindingException("Failed to initialize cursor", e);
 }
}
```

getDataObject

```
public DataObject getDataObject() throws DataBindingException
```

This method builds an instance of SDO with the data that is returned from the backend application. For example, when an adapter executes a Retrieve operation, the data returned from the backend application is held in the adapter structured record implementation. In this method the adapter reads data from the backend application and builds and SDO instance.

To perform this task the adapter should use DESPI APIs. Initialize the record with initializeOutput(), then call getNext() to build data in SDO.

This method should take care of building an instance of BG is the methods getNamespaceURI()and getBusinessObjectName() return a type BG.

For operations where getNext() should be invoked multiple times like RetrieveAll, the databinding should call getNext() multiple times add the built BusinessObject to the list of BusinessObjects within the Container BO.

getRecord

```
public Record getRecord() throws DataBindingException
```

This method should return the instance of record being build in setDataObject() call or passed through setRecord() call.

setRecord

```
public void setRecord(Record arg0) throws DataBindingException
```

This method should hold the instance of the record passed to this method as a record instance for the binding implementation. This record instance is used in method getDataObject().

Abstract methods (getNamespaceURI and getBusinessObjectName)

```
public abstract String getNamespaceURI();
public abstract String getBusinessObjectName();
```

DataBinding implementation uses these two methods in getDataObject() call to determine the SDO type that the binding should instantiate.

The generated databinding classes described the section *DataBinding generator* provide the implementation for the abstract methods.

DataBinding generator

To enable the right business object type being made available to DataBinding implementations adapters should implement `DataBindingGenerator` interface.

Adapters should implement `com.ibm.j2ca.extension.databinding`, `WBIDataBindingGenerator` and implement provide a default constructor implementation.

Call the super class constructor and pass in the name of the adapter and the absolute classname of the base DataBinding implementation.

Sample code:

```
public TwineBallDataBindingGenerator() {
  super("TwineBall", "com.ibm.j2ca.sample.twineball.emd.runtime.TwineBallDataBinding");
 }
```

EMD implementations should set the absolute name of the class for `DataBindingGenerator` as the generator classname in DataDescription instance. Set the DataBinding Classname to null.

Bidirectional language support

Specifying bidirectional properties allows your adapter to exchange data in a variety of bidirectional formats.

Bidi normalization is supported the following fashion:

- When WBIStructuredRecord is initialized with data that contains Bidi annotations, Cursors and accessors that are associated with that structured record will automatically translate the content into or from the Bidi format specified in the annotations, via special cursors and accessors that wrap the versions provided by the data exchange factory .

Problem determination

You can implement messages to accompany a range events.

Fault handling support

Through enhancements to the Service Component Architecture (SCA) and the Adapter Foundation Classes (AFC), the WebSphere Adapter Toolkit provides fault handling support. Fault handling allows the developer to differentiate information technology (IT) exceptions from business processing exceptions during outbound processing.

A fault is an exception condition that alters the normal flow of a business process. Typically, a fault represents a predictable error that has a well-defined action. Presenting errors as faults instead of exceptions makes it easier for you to configure recovery processing, because fault handling does not require you to write Java code to catch and handle an exception. Adapters created with the WebSphere Adapter Toolkit generate several faults.

Fault handling support adheres to the DataBinding model, in that when an exception is thrown by an adapter, a fault selector determines that it is a fault and the fault (data) binding converts the exception to a fault business object and returns it to the server runtime.

Fault artifacts:

To facilitate fault handling support, WebSphere Adapter Toolkit provides artifacts that include fault classes, fault business objects, a fault selector, and a fault binding class.

The following table shows AFC fault classes, corresponding fault business names and fault type names. The configured fault binding and a descriptive example of each fault can be found in the section about configuration for fault handling.

Fault Exception in AFC	Corresponding Fault Name	Corresponding Fault Type Name
DuplicateRecordException	DUPLICATE_RECORD	DuplicateRecordFault
InvalidRequestException	INVALID_REQUEST	InvalidRequestFault
MatchesExceededLimitException	MATCHES_EXCEEDED_LIMIT	MatchesExceededLimitFault
MissingDataException	MISSING_DATA	MissingDataFault
MultipleMatchingRecordsException	MULTIPLE_MATCHING_RECORDS	MultipleMatchingRecordsFault
RecordNotFoundException	RECORD_NOT_FOUND	RecordNotFoundFault

In addition to the fault classes, the following fault selector class and base fault binding class are provided:
- WBIFaultSelectorImpl
- WBIFaultDataBindingImpl

A utility class named `FaultBOUtil` can help you define simple custom fault business objects.

How to support fault handling:

Understand the following concepts for implementing fault handling into your adapter.

Before you define faults, review adapter processing to determine which error conditions can be categorized as faults, rather than as exceptions. You will likely be able to apply at least one of the faults provided in the Adapter Foundation Classes. Some unique conditions might be categorized as faults, but fault classes in the Adapter Foundation Classes might not be provided for the conditions. Error conditions from which you can recover might be candidates for fault handling. IT exceptions, such as one time configuration issues (wrong password, incorrect directory permissions, and so on) are not candidates for fault handling.

When naming a fault, ensure that the name describes the condition and is independent of the technology or adapter you are using. For example, the fault class name for an *SAP IDoc record not found* condition should be `RecordNotFoundFault`, not `SAPIDOCRecordNotfoundFault`. Additionally, you might not need to define new faults if the same semantic meaning can apply to multiple conditions (for example, *RecordNotFoundFault*, *FileNotFoundFault*, and *ObjectNotfoundFault*).

Implementing faults:

As part of the external service discovery process, you generate fault business objects and create the method that copies them to your outbound module.

The following examples show these fault exceptions:
- DuplicateRecord
- MatchesExceedLimit

Note: To implement fault handling in the adapter, add getXMLListFunctions method and getBFFunctions method in the extension to WBIOutboundFunctionDescriptionImpl.

Modifying getXMLListFunctions method

Modifying getXMLListFunctions involves the following changes:
- Adding a line to invoke a new method.
- Creating the new method or add some lines to an existing method to specify the faults
- Completing the stub to implement the FaultDataDescription class

The following example shows the first two changes:

```
private void getXMLListFunctions(ArrayList functionDescriptions, PropertyGroup pg,
String relativePath, JDEXMLListMetadataObject metadataObj,
WBIMetadataImportConfigurationImpl spec) throws MetadataException{
...
  while(iterator.hasNext()) {
   String operation = JDEESDConstants.RETRIEVEALL; /* this is only operation
supported for XML Lists */


   WBIOutboundFunctionDescriptionImpl funcDesc = new WBIOutboundFunctionDescriptionImpl();
   JDEXMLListQueryDataDescription queryDataDesc =(JDEXMLListQueryDataDescription)iterator.next();
            //Added for Faults
   addFaultsToXMLListDataDescriptionBasedOnOperationName(funcDesc,operation);
```

```
            //Added for Faults   funcDesc.setName(operation.toLowerCase() + queryDataDesc.getBOName());
   getLogUtils().trace(LogLevel.FINEST,
CLASSNAME, "getXMLListFunctions", "Setting input data description
to: " + queryDataDesc.getName().toString() + " for function: "
+ funcDesc.getName()); //$NON-NLS-1$
   funcDesc.setInputDataDescription(queryDataDesc);
   funcDesc.setOutputDataDescription(containerDataDesc);

   JDEInteractionSpec iSpec = new JDEInteractionSpec();
   WBISingleValuedPropertyImpl maxCount = (WBISingleValuedPropertyImpl)pg.getProperty(JDEESDProperties.MAXRECORDS);
   if (maxCount.getValue() == null)
    iSpec.setMaxRecords(((Integer) maxCount.getPropertyType().getDefaultValue()).intValue());
   else
    iSpec.setMaxRecords(((Integer) maxCount.getValue()).intValue());
   WBISingleValuedPropertyImpl timeoutProp = (WBISingleValuedPropertyImpl)pg.getProperty(JDEESDProperties.ISPECTIMEOUT);
   if (timeoutProp.getValue() != null && ((Integer) timeoutProp.getValue()).intValue()>0){
    iSpec.setTimeout(((Integer) timeoutProp.getValue()).intValue());
   }

   iSpec.setFunctionName(operation);
   funcDesc.setInteractionSpec(iSpec);
   functionDescriptions.add(funcDesc);
  }
...
}

 private void addFaultsToXMLListDataDescriptionBasedOnOperationName(
   WBIOutboundFunctionDescriptionImpl funcDesc, String operationName)
   throws MetadataException {
  // Define XSDs for faults supported by this adapter.
  try {
   // During implementation - add faults based on the operationName
   // parameter. For ex an operation may have more than one fault.
   // And so the below logic will have to be using conditions
   // if(operationName.equals("CREATE")) {} then do this etc.

   JDEXMLListFaultDataDescription fdesc1 = new JDEXMLListFaultDataDescription();
   JDEXMLListFaultDataDescription fdesc2 = new JDEXMLListFaultDataDescription();

   BusinessObjectDefinition bo = FaultBOUtil.createDuplicateRecordBO();
   URI uri = new URI("./" + FaultBOUtil.DUPLICATE_RECORD_NAME //$NON-NLS-1$
     + EMDConstants.XSD);
   fdesc1.put(FaultBOUtil.FAULT_TARGET_NS, uri, bo.serialize());
   fdesc1.setGenericDataBindingClassName("com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl");
   fdesc1.setFaultName(FaultBOUtil.DUPLICATE_RECORD_NAME);

   bo = FaultBOUtil.createMatchesExceededLimitBO();
   uri = new URI("./" + FaultBOUtil.MATCHES_EXCEEDED_LIMIT_NAME //$NON-NLS-1$
     + EMDConstants.XSD);
   fdesc2.put(FaultBOUtil.FAULT_TARGET_NS, uri, bo.serialize());
   fdesc2.setGenericDataBindingClassName("com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl");
   fdesc2.setFaultName(FaultBOUtil.MATCHES_EXCEEDED_LIMIT_NAME);

   FaultDataDescription desc[] = new FaultDataDescription[] { fdesc1,
     fdesc2 };

   funcDesc.setFaultSelectorClassname("com.ibm.j2ca.extension.emd.runtime.WBIFaultSelectorImpl");
   funcDesc.setFaultDataDescriptions(desc);

  } catch (Exception e) {
   throw new MetadataException(
     "Unable to create fault BO definitions " + e.getMessage(), e); //$NON-NLS-1$
  }
 }
```

The following example shows code for implementing FaultDataDescription in the JDEXMLListFaultDataDescription class:

```
public class JDEXMLListFaultDataDescription implements
  FaultDataDescription {

 public JDEXMLListFaultDataDescription() {
  super();
  // TODO Auto-generated constructor stub
 }

 private String faultName = null;

 public String getFaultName() {
  // TODO Auto-generated method stub
  return faultName;
 }

 public void setFaultName(String faultName) {
  this.faultName = faultName;
 }
}
```

Modifying getBFFunctions method

Modifying getBFFunctions involves the following changes:

- Adding a line to invoke a new method
- Creating the new method or add some lines to an existing method to specify the faults
- Completing the stub to implement the FaultDataDescription class

The following example shows the first two changes:

```
private void getBFFunctions(ArrayList functionDescriptions, PropertyGroup pg,
String relativePath, ArrayList dataDescriptions) throws MetadataException {
  Iterator iterator = dataDescriptions.iterator();
  while(iterator.hasNext()){
   JDEBFContainerDataDescription dataDesc = (JDEBFContainerDataDescription)iterator.next();
   JDEBFOperationASI[] ops = dataDesc.getOperationASI();
   ArrayList operations = new ArrayList();
   for(int i=0; i<ops.length; i++) {
    if(ops[i].getBsfnNames().length>0){
     operations.add(ops[i].getName());
    }
   }
   Iterator opIterator = operations.iterator();
   while (opIterator.hasNext())
   {
    String operation = (String)opIterator.next();
    WBIOutboundFunctionDescriptionImpl funcDesc = new WBIOutboundFunctionDescriptionImpl();
       //Added for Faults
    addFaultsToBFDataDescriptionBasedOnOperationName(funcDesc,operation);
             //Added for Faults    funcDesc.setName(operation.toLowerCase() + dataDesc.getBOName());
    funcDesc.setInputDataDescription(dataDesc);
    funcDesc.setOutputDataDescription(dataDesc);

    JDEInteractionSpec iSpec = new JDEInteractionSpec();
    iSpec.setFunctionName(operation);
    funcDesc.setInteractionSpec(iSpec);
    functionDescriptions.add(funcDesc);

   }
  }
 }
 private void addFaultsToBFDataDescriptionBasedOnOperationName(WBIOutboundFunctionDescriptionImpl funcDesc, String operationName)
   throws MetadataException {
  // Define XSDs for faults supported by this adapter.
  try {
   //During implementation - add faults based on the operationName parameter. For ex an operation may have more than one fault.
   //And so the below logic will have to be using conditions if(operationName.equals("CREATE")) {} then do this etc.

   JDEBFFaultDataDescription fdesc1 = new JDEBFFaultDataDescription();
   JDEBFFaultDataDescription fdesc2 = new JDEBFFaultDataDescription();

   BusinessObjectDefinition bo = FaultBOUtil.createDuplicateRecordBO();
   URI uri = new URI("./" + FaultBOUtil.DUPLICATE_RECORD_NAME //$NON-NLS-1$
     + EMDConstants.XSD);
   fdesc1.put(FaultBOUtil.FAULT_TARGET_NS, uri, bo.serialize());
   fdesc1.setGenericDataBindingClassName("com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl");
   fdesc1.setFaultName(FaultBOUtil.DUPLICATE_RECORD_NAME);


   bo = FaultBOUtil.createMatchesExceededLimitBO();
   uri = new URI("./" + FaultBOUtil.MATCHES_EXCEEDED_LIMIT_NAME //$NON-NLS-1$
     + EMDConstants.XSD);
   fdesc2.put(FaultBOUtil.FAULT_TARGET_NS, uri, bo.serialize());
```

```
    fdesc2.setGenericDataBindingClassName("com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl");
    fdesc2.setFaultName(FaultBOUtil.MATCHES_EXCEEDED_LIMIT_NAME);

    FaultDataDescription desc[] = new FaultDataDescription[] {fdesc1, fdesc2};

    funcDesc.setFaultSelectorClassname("com.ibm.j2ca.extension.emd.runtime.WBIFaultSelectorImpl");
    funcDesc.setFaultDataDescriptions(desc);

   } catch (Exception e) {
    throw new MetadataException(
      "Unable to create fault BO definitions " + e.getMessage(), e); //$NON-NLS-1$
   }
  }
```

The following example shows code for implementing FaultDataDescription in the JDEBFFaultDataDescription class:

```
public class JDEBFFaultDataDescription implements
  FaultDataDescription {

 public JDEBFFaultDataDescription() {
  super();
  // TODO Auto-generated constructor stub
 }

 private String faultName = null;

 public String getFaultName() {
  // TODO Auto-generated method stub
  return faultName;
 }

 public void setFaultName(String faultName) {
  this.faultName = faultName;
 }
}
```

Configuration for fault handling:

Adapter Foundation Classes fault names and the corresponding fault binding names are used in the fault configuration.

A fault name is defined within each fault class. The base fault binding is configured unless attributes are unique.

The following table includes examples of situations when an adapter might throw each type of fault. These are *examples only*.

Table 4. Fault name and configured fault binding

Fault Name	Configured Fault Binding
DUPLICATE_RECORD	`com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl` The adapter throws this fault when processing an outbound Create operation when an error occurs because the specified file already exists in the specified directory path.

Table 4. Fault name and configured fault binding (continued)

Fault Name	**Configured Fault Binding**
INVALID_REQUEST	`com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl` When input to the operation does not have the required characteristics, the adapter throws this fault. Specific errors that can result include the following: • For a Create operation: – A ChangeSummary is provided but the required child business objects are not marked as created (per strict conventions) – A business object is marked as deleted in the ChangeSummary (assertion optional) – The input business object specifies key values, but the server supports only auto-creation – The input business object does not contain key values, but the server requires them • For a Delete operation: – A ChangeSummary is provided but the required child business objects are not marked as deleted (per strict conventions) – A business object is marked as created in the ChangeSummary (assertion optional)
MATCHES_EXCEEDED_LIMIT	`com.ibm.j2ca.extension.emd.runtime.MatchingFaultDataBinding` When processing the processing of an RetrieveAll operation, the adapter throws this fault if the number of records returned from a database query exceed the *maximum number of records* property in the interaction specification.
MISSING_DATA	`com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl` If the business object that is passed to the outbound operation does not have all the required attributes, the adapter throws this fault.
MULTIPLE_MATCHING_RECORDS	`com.ibm.j2ca.extension.emd.runtime.MatchingFaultDataBinding` When processing a Retrieve operation, the adapter throws this fault if the query returns more than one record for the specified keys.
RECORD_NOT_FOUND	`com.ibm.j2ca.extension.emd.runtime.WBIFaultDataBindingImpl` When processing a data retrieval operation, the adapter throws this fault if the record is not found in the database for the keys specified. This fault can occur for the Delete, Update, RetrieveAll, and RetrieveAll operations.

Defining custom faults:

You can define custom faults for fault handling.

- Define the fault class

  Implement BaseFaultException and define additional attributes if necessary. In defining the BaseFaultException class you can see the convention for specifying the fault name. For example, the RecordNotFoundException fault name is RECORD_NOT_FOUND.
- Define a fault binding

  Only required if you have defined additional attributes for your fault.
- Define the fault business object

You can use the FaultBOUtil to define the fault business object, as long as either no attributes or only simple attributes are added. This should amount to a few lines of code, See Implementing Faults for an example.

Note: The model for fault classes and fault business objects is a 1-to-1 relationship, the base fault business object cannot be used even if no additional attributes are needed. This is because SCA does not pass back the fault name to the client / server runtime. Instead, the fault name is resolved to the fault (BO) type. So if you reuse fault BOs, you cannot definitively determine which fault occurred.

Logging and tracing messages

Providing information about the runtime state of the adapter is a critical aspect of adapter development. The Adapter Foundation Classes include the `LogUtils` class. When implemented, this functionality enables developers to target information to a variety of user roles, enabling them to filter information by levels of importance and to generate informational events that can be monitored by and acted upon by WebSphere Process Server.

Information about the runtime state of the adapter is invaluable not only to support teams trying to resolve problems but also to users looking to monitor the adapter and track its operations. For these reasons, you should focus early in your adapter development process on what information to provide, which users to target, and how to most efficiently and clearly communicate the information.

The JCA 1.5 specification provides minimal support for communicating information to users. It defines a single stream to which the adapter writes any and all information. As a result, without additional tools or support, users cannot filter information, analyze information or, in general, easily determine what information is of interest to them.

The New Adapter wizard generates code skeleton using Adapter Foundation Classes to provide you with a consistent method to get the `LogUtils` object in your own implementation. This is very useful when an adapter wants to record business information or needs to track execution flow. The `LogUtils` class allows you to direct information usefully to a variety of users by generating three types of ″messages″: trace, log, and event messages. Each has a distinct purpose and conveys different information.

- The following guidelines apply to trace messages:
  - They do not contain information needed by general users to resolve problems
  - They encapsulate information intended for support and other development teams
  - They need not be translated
  - They can be hard-coded in the adapter itself
  - They feature three levels of detail: fine, finer, and finest
- The following guidelines apply to log messages:
  - They encapsulate information such as warnings and errors that are targeted at general users of the adapter
  - Rather than hard-code log messages in the adapter, place them in a separate log message file to facilitate translation into localized languages
  - They feature multiple levels that users can employ to filter log messages
- The following guidelines apply to event messages:

– They provide information on the state of the adapter for use by monitor tooling
– Represented as common base event data at run time, event messages can be included in the extended `LogUtils.log` method signatures

Support for protecting sensitive user data in log and trace files

WebSphere Adapter Toolkit provides support for confidential tracing of properties. This means that when you decide whether a property might contain potentially sensitive data and the use a special interface when you record that information in a log or trace message. Data recorded with the special interface is displayed in the logs by a ″XXXX″. This functionality is most useful to customers who mainly deal with a lot of confidential information such as banks, healthcare companies and defense. This property is a part of the Adapter Foundation Classes, and so it can be used by any adapters.

When a property is marked as confidential and if it needs to be logged or traced, then you record the information using a special `confidential` log and trace method provided in the `logUtils` will be invoked.

The following types of information are considered potentially sensitive data and will be hidden:
- The contents of a business object
- The contents of the object key of the event record
- User names , Password, Environment and Role
- The URL used to connect to the

The following types of information are not considered potentially sensitive data and will not be hidden:
- The contents of the event record that are not part of the event record object key. For example, the `transactionID` (XID), event ID, business object name, and event status
- Business object schemas
- Call sequences

Inserting log and trace messages

The WebSphere Adapter Toolkit automatically provides entry and exit tracing statements to the generated code, excluding constructors and assessor methods. The WebSphere Adapter Toolkit does not add logging and tracing statements to the generated code for a JCA resource adapter. You will not be able to manually add logging and tracing to the generated JCA adapter code.

You can insert log and/or trace statements into your generated IBM WebSphere adapter code using a dialog box that collects information about the log or trace statement to be generated and insert the appropriate code at the cursor position.

Trace messages:

Trace messages convey information that is intended for support teams and developers. Such information includes stack dumps for exceptions and operation logic for debugging purposes. Because trace messages are directed at the teams that wrote or support the adapter rather than customers, trace messages need not be translated and can, in fact, be hard-coded in the adapter.

Writing a trace message

You use the trace method of the LogUtils class to generate a trace message. This method has two signatures. One of them is informational. The other is associated with an exception.

```
void trace
   (Level l, String classname, String method, String msg)

void trace
   (Level l, String classname, String method, String msg, Exception ex)
```

In an outbound or inbound scenario, to get the LogUtils object instance, in WebSphere Integration Developer, Right click on the method and select the following option:

- **Source → Insert Trace Statement**

In an outbound scenario, you can use the logger property as a consistent way to get a log object to write log or trace messages into log or trace files.

Trace messages might contain data from the customer's EIS. Because a customer might be unwilling to send a trace file that contains sensitive data to a support specialist for analysis, you can optionally use the traceConfidential method of the LogUtils class to write confidential trace messages whose EIS-specific content is replaced by a string of XXX's when the customer enables the HideConfidentialTrace property. Like the trace method, the traceConfidential method has two signatures. One of them is informational. The other is associated with an exception.

```
void traceConfidential
   (Level l, String classname, String method, String msg, Object[] confidentialData)

void traceConfidential
   (Level l, String classname, String method, String msg, Object[] confidentialData, Exception e)
```

Example of informational trace message

```
getLogUtils().trace
   (Level.Fine, "FooAdapter", "openConnect", "Successfully connected to Foo server");
```

Example of exception trace message

```
getLogUtils().trace(Level.Finer, "FooAdapter", "closeConnection",
  "While attempting to close the connection to Foo server,
  Foo API reported that the server had already closed the connection
  previously due to inactivity.  Ignoring because connection was closed all the same",
fooAPIException);
```

Example of trace message for the outbound scenario

```
public HelloWorldConnectionFactory(ConnectionManager connMgr, WBIManagedConnectionFactory mcf)
{
  super(connMgr, mcf);
    getLogUtils().trace(Level.FINE,"com.ibm.helloworld.outbound.HelloWorldConnectionFactory",
        "HelloWorldConnectionFactory()", "test");
 }
```

Example of trace message for the inbound scenario

```
public javax.resource.cci.Record getRecordForEvent(com.ibm.j2ca.extension.eventmanagement.Event event)
   throws javax.resource.ResourceException,
   javax.resource.spi.CommException {
     logger.trace(Level.FINE,
    "com.ibm.helloworld.inbound.HelloWorldEventStoreWithXid",
```

```
    "getRecordForEvent()", "test");
  return null;
 }
 }
```

Example of trace message for the outbound scenario with confidential tracing property enabled

```
public HelloWorldConnectionFactory(ConnectionManager connMgr, WBIManagedConnectionFactory mcf) {
  super(connMgr, mcf);
   getLogUtils().traceConfidential(Level.FINE,
     "com.ibm.helloworld.outbound.HelloWorldConnectionFactory",
     "HelloWorldConnectionFactory()", "test",
     new Object[] { "str" });
 }
```

Example of trace message for the inbound scenario with confidential tracing property enabled

```
public javax.resource.cci.Record getRecordForEvent(com.ibm.j2ca.extension.eventmanagement.Event event)
   throws javax.resource.ResourceException,
   javax.resource.spi.CommException {
  logger.traceConfidential(Level.FINE,
    "com.ibm.helloworld.inbound.HelloWorldEventStoreWithXid",
    "getRecordForEvent()", "test", new Object[] { "str" });
  return null;
 }
```

Note: In the previous code snippets, getLogUtils() and logger are instances of the LogUtils class.

Trace levels

Three trace levels allow users to adjust the level of detail. Consult the guidelines shown in the following table to determine which trace level to assign to a trace message.

Table 5. Trace level indicators

Level	Indicator	Significance
Fine	1	This trace features the lowest level of detail. It includes broad actions taken by the adapter such as establishing a connection to the EIS, converting an event in the EIS to a business object (key values only), processing a business object (key values only).
Finer	2	This trace provides more detailed information on adapter logic, including API calls to the EIS and any parameters or return values.
Finest	3	This is the most detailed level and includes method entry, exit, and return values. Include complete business object dumps and all detail needed to debug problems.

Configuring log and trace detail level

Set the trace level on the package that you want to trace to `all` in the **Change Log Detail Levels** field: For example, if the adapter package name is **com.ibm.myadapter**, modify the **Change Log Detail Levels** pane and add `com.ibm.myadapter.* = all.`"

Performance considerations

Tracing assists developers and troubleshooters. Due to the significant performance cost incurred by tracing, however, many customers disable it in production environments. When developing or troubleshooting, it is good practice to check whether tracing is enabled before building to generate trace messages. You can do this by checking method `LogUtils.isTraceEnabled(`*`java.util.logging.Level`*`)` before building the adapter. The following is an example:

```
if(logUtils.isTraceEnabled(Level.Fine) {
    getLogUtils().trace(...);
```

Log messages:

Log messages convey timely information intended for consumption by customers: warnings about potential problems, errors that have occurred and suggested fixes for those errors, and information that is necessary or helpful to understanding how the adapter operates.

Message files

To facilitate translation of log messages for different user groups, place all log messages in a resource bundle file rather than hard-code them in the adapter itself. The convention for packaging this bundle file is to embed it in the adapter RAR as `<adapter package>.emd/logMessages.properties`.

The message file can contain one or more messages. Each message should be comprised of three parts:

- **Message Identifier –** The message ID follows the format *NNNNNmmmmS*, where *NNNNN* is a five-letter component prefix, *mmmm* is the message number, and *S* identifies the message type. For example, a message identifier such as `ABCDE0001E` has a component identifier of `ABCDE` and a message number of `0001`. The message type `E` tells you this is an error message. The component identifier must be registered with IBM to avoid conflicts between products. The type identifier should conform to one of the values specified in the Log Level table shown below. The message number is left to you.
- **Explanation –** The explanation provides an in-depth description of the message. Assume that the customer is unfamiliar with the meaning of the base message. The explanation is the first level of help documentation for users, not a crutch for a poorly written base message.
- **User Action –** For every explanation of what went wrong, there usually are actions that customers can take to rectify the situation or to ensure that it doesn't happen again. The User Action field provides detailed information on what customers can do and is very much like first-level help documentation.

The following is an example of a message file:

```
0001=CWYBS0001I: Adapter {1} started.
0001.explanation=The adapter has successfully initialized and is now ready to service requests.
0001.useraction=

0004=CWYBS9999E: Failed to establish connection link to server on host {1}.
0004.explanation=The adapter is unable to contact the backend application.
  Business data cannot be exchanged with the backend until this issue is resolved.
0004.useraction= Check your adapter configuration and ensure that the
  specified host and port match the machine and port on which the backend is listening.
  If correct, check that your backend application is on-line and accepting requests.
```

Message types

There are two message types for adapters, `ADAPTER_RBUNDLE`. The `BASE_RBUNDLE` is reserved for the Adapter Foundation Classes. The message types are used to distinguish between the adapter and the base classes message file. The default value of Message Type field is `ADAPTER_RBUNDLE`. You use only the `ADAPTER_RBUNDLE` message type because you should only access adapter message file.

Log levels

Log Levels and indicators

Level	Indicator	Significance
Fatal	F	Task cannot continue. Component cannot function
Severe	E	Task cannot continue. Component can still function. This can also indicate an impending fatal error, including situations that strongly suggest that resources are on the verge of being depleted.
Warning	W	Potential error or impending error. This includes conditions that indicate a progressive failure - for example, the potential leaking of resources.
Audit	A	Significant event affecting server state or resources
Info	I	General information outlining overall task progress.

Writing a log message

Use the log method of the `LogUtils` class to generate log messages. This method accepts parameters similar to that of the trace file. Instead of providing a hard-coded message, however, provide a key to the message from the log message file. This log method can also take optional parameters if there are values to be substituted in the message.

```
void log
  (Level l, int bundleType, String classname, String method, String msgKey)

void log
  (Level l, int bundleType , String classname, String method, String msgKey,
   Object[] params)
```

For example, if you have defined parameters in your log message such as `Successfully processed business object {1} with id {2}`, you would provide values for those parameters using the argument `params[]`.

Note: Since parameters are not translated, avoid passing hard-coded phrases as parameters because the resulting message will be a combination of English and

translated text. Values that are language-independent, such as key values or object names, are appropriate as log message parameters.

Similarly to trace messages, parameters in a log message can contain data from the customer's EIS. Because a customer might be unwilling to send a log file that contains sensitive data to a support specialist for analysis, you can optionally use the logConfidential method of the `LogUtils` class to generate confidential log messages whose EIS-specific content is replaced by a string of XXX's when the customer enables the HideConfidentialTrace property. Like the log method, the logConfidential method accepts parameters similar to that of the trace file. Instead of providing a hard-coded message, however, provide a key to the message from the log message file. The logConfidential method can also take optional parameters if there are values to be substituted in the message.

```
void logConfidential
  (Level l, int bundleType, String classname, String method, String msgKey, Object[] params)

void logConfidential
  (Level l, String classname, String method, String msgKey, CBEEngineData engine)

void logConfidential
  (Level l, String classname, String method, String msgKey, Object[] params)

void logConfidential
  (Level l, String classname, String method,String msgKey, Object[] params, CBEEngineData engineData)
```

In an outbound or inbound scenario, to get the `LogUtils` object instance, in WebSphere Integration Developer, Right click on the method and select the following option:

- **Source → Insert Log Statement**

.

Example of log message for the outbound scenario

```
public HelloWorldConnectionFactory(ConnectionManager connMgr, WBIManagedConnectionFactory mcf) {
  super(connMgr, mcf);
  getLogUtils().log(Level.INFO, LogUtilConstants.ADAPTER_RBUNDLE,
    "com.ibm.helloworld.outbound.HelloWorldConnectionFactory",
    "HelloWorldConnectionFactory()", "10");
       }
```

Example of log message for the inbound scenario

```
public javax.resource.cci.Record getRecordForEvent(com.ibm.j2ca.extension.eventmanagement.Event event)
   throws javax.resource.ResourceException,
   javax.resource.spi.CommException {

  logger.log(Level.INFO, LogUtilConstants.ADAPTER_RBUNDLE,
    "com.ibm.helloworld.inbound.HelloWorldEventStoreWithXid",
    "getRecordForEvent()", "10");
  return null;
 }
```

Example of log message for the outbound scenario with confidential tracing property enabled

```
public HelloWorldConnectionFactory(ConnectionManager connMgr, WBIManagedConnectionFactory mcf) {
  super(connMgr, mcf);
  getLogUtils().logConfidential(Level.INFO,
    LogUtilConstants.ADAPTER_RBUNDLE,
    "com.ibm.helloworld.outbound.HelloWorldConnectionFactory",
    "HelloWorldConnectionFactory()", "10", new Object[] { "str" });
 }
```

Example of log message for the inbound scenario with confidential tracing property enabled

```
public javax.resource.cci.Record getRecordForEvent(com.ibm.j2ca.extension.eventmanagement.Event event)
   throws javax.resource.ResourceException,
   javax.resource.spi.CommException {
  logger.logConfidential(Level.INFO, LogUtilConstants.ADAPTER_RBUNDLE,
    "com.ibm.helloworld.inbound.HelloWorldEventStoreWithXid",
    "getRecordForEvent()", "10", new Object[] { "test" });
     return null;
 }
```

Note: In the previous code snippets, getLogUtils() and logger are instances of the LogUtils class.

Monitoring and measuring performance

The purpose of monitoring is to observe the progress of execution of WebSphere Business Integration applications, and the WebSphere Business Integration system itself, and publish the results of this observation.

Monitoring can be accomplished by:

- Using the Common Event Infrastructure (CEI), a set of modular event processing components that provide functions to capture information about significant system or business occurrences.
- Using the Performance monitoring infrastructure (PMI) to collect data, such as average response time and total number of requests, from various components in a server runtime environment, and organizes the data into a tree structure.
- Using Application response measurement (ARM) to monitor the availability and performance of applications.

Common Event Infrastructure (CEI):

The Common Event Infrastructure (CEI) is a set of modular event processing components that provide functions to capture information about significant system or business occurrences.

WebSphere Process Server includes the Common Event Infrastructure technology, which adapters use to create, transmit, persist and distribute events.

Note: If an adapter is running on a broker that does not use the IBM CEI technology but instead uses its own event monitoring technology; that broker can also plugin its monitoring infrastructure with the adapters by implementing the interfaces described in the sections that follow and by optionally using schema definitions (.xsd and .mes files).

EventSourceContext

The `EventSourceContext` interface provides the context for a monitored component and is the starting point for the adapters. The `EventSourceContext` interface provides APIs to obtain *event source*, which is an application or component that submits an event creation request to CEI. Each event source defines a set of event points, which represent the points where CEI events are triggered.

```
/**
 * Provides the context for a monitored component.
 */

package com.ibm.j2ca.extension.monitoring.CEI;
```

```
public interface EventSourceContext
{
 /**
   * Returns an event source for a monitored element.
   * @param elementKind an artifact kind that can be monitored e.g ResourceAdapter.
   * @param elementName the name of the monitored element
   * @return the event source object that encapsulates the element to be monitored
   */

    EventSource getEventSource(String elementKind, String elementName);

 /**
   * Creates an event source for a monitored element. The usage is similar to
   * java.util.logging.Logger.
   * @param componentTypeQName : The element type can be specified using the element
   * type from
   *   a schema which defines the structure/syntax of the artifact itself
   *    e.g.http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
   * @param componentQName  the name of the component ,
   *      e.g  http://www.ibm.com/j2ca/ResourceAdapter:Customer"
   * @return the event source factory for the component to be monitored
   */

    public interface Factory{
     EventSourceContext create(String componentTypeQName, String componentQName);
    }
}
```

EventSource

Event Sources are applications or components that submit event creation requests to CEI. Each *monitorable* element defines an event source an example of event source is an adapter. An event source is used to retrieve event points in order to send monitoring events to a CEI logger.

```
package com.ibm.j2ca.extension.monitoring.CEI;

/**
 *      An event source represents a monitoarable element kind such as an adapter....
 *      Each monitorable element defines an event source,  Each event source defines a
 *      set of component-element specific event points.  An event source object is used
 *      to retrieve event points to fire monitoring events
 */
public interface EventSource
{
     /**
     * returns an EventPoint for the monitored element
     * @param eventPointName a valid event nature for this event
     */
    public EventPoint getEventPoint(String eventPointName);
}
```

EventPoint

Every monitorable component needs to define the event points. Each event point defines an event and the data or payload associated with that event. The EventPoint is used to fire monitoring events. The client of an event point needs to know the payload of the fired events. Where the payload of an event can be specified in an event catalog for each component.

```
package com.ibm.j2ca.extension.monitoring.CEI;
/**
 * Every monitorable component needs to defines the event points.  Each event point
 * defines an event and the data/payload associated with that event. The EventPoint
 * is used to fire monitoring events.
```

```
 * The client of an event point needs to know the payload of the  fired events.
 */
public interface EventPoint
{
   /**
 *return the name of the event point
 */

    String getName();

   /**
 * Checks if an event needs to be fired for this event point. This method minimizes the
 *  overhead of inactive monitoring
 * points.   returns: true if this event point fires monitoring events.
    */

    boolean isEnabled();

   /**
 * Fires a monitoring event.
 * @param name  the name of the payload data element.
 * @param value the value of the payload data element
 */

    void fire(String name, Object value);

   /**
    *  Fires a monitoring event,
    *  it is a convenient method for payloads with two data elements.
*/

    void fire(String firstName, String secondName, Object firstValue, Object secondValue);

 /**
     * Fires a monitoring event
     * It is a convenient method for payloads with list of data elements.
     */
    void fire(String[] names, Object[] values);
}
```

Unique Id

The Unique Id interface can be used to uniquely identify event points:

```
/**
 * Every monitorable component needs to defines the event points.
 * Each event point defines
 * an event and the data/payload associated with that event. The EventPoint is
 * used to fire monitoring events.
 * The client of an event point needs to know the payload of the events.h
 */

public interface AdapterContext
{
 public String getUniqueId();

}
```

Example of the schema definition files

1. **Monitorable element schema (.mes file):**

   Monitorable element schema can be used to define element types that can be logged into the CEI database (Polling, InboundEventDelivery, Outbound etc) and it can also define natures that are available for each element type (entry, exit, failed, polling etc).

```
<?xml version="1.0" encoding="UTF-8"?>
<EventNaturesSpec name="EventNatures" targetNamespace="http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
 xmlns="http://www.ibm.com/xmlns/prod/websphere/monitoring/6.1/mes"
 xmlns:eis="http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter" shortName="ResourceAdapter">

 <Property>CEI</Property>
<ElementKind name="Polling">
  <EventNature name="STARTED" eventName="eis:WBI.JCAAdapter.Polling.STARTED" />
  <EventNature name="STOPPED" eventName="eis:WBI.JCAAdapter.Polling.STOPPED" />
 </ElementKind>
</EventNaturesSpec>
```

2. **Xsd schema (.xsd):**

   Xsd schema can be used to provide CEI specific of each data elements and it also defines the types of events that can be emitted for the data elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
 xmlns:eis="http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
 xmlns:wbi="http://www.ibm.com/xmlns/prod/websphere/monitoring/6.1">
 <import namespace="http://www.ibm.com/xmlns/prod/websphere/monitoring/6.1" schemaLocation="WBIEvent.xsd" />
 <complexType name="WBI.JCAAdapter.Polling.STARTED">
  <complexContent>
   <extension base="wbi:WBIMonitoringEvent">
    <sequence>
     <element name="PollFrequency" type="int" minOccurs="1" maxOccurs="1" />
     <element name="PollQuantity" type="int" minOccurs="1" maxOccurs="1" />
    </sequence>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="WBI.JCAAdapter.Polling.STOPPED">
  <complexContent>
   <extension base="wbi:WBIMonitoringEvent"></extension>
  </complexContent>
 </complexType>

</schema>
```

Extending Common Event Infrastructure logging on WebSphere Process Server:

You can extend CEI logging to WebSphere Process Server by adding custom events.

Example of how to log an event

The following example describes how to log an event named *Polling*, when the event action is `Started` and you want to log two integers values for this event in the CEI database.

1. Monitorable Element Schema ".mes" file changes

   Defines element types that you want to monitor (Polling for example) and it also defines natures that are available for each element type (STARTED).

```
<?xml version="1.0" encoding="UTF-8"?>
<EventNaturesSpec name="EventNatures" targetNamespace=
"http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
 xmlns="http://www.ibm.com/xmlns/prod/websphere/monitoring/6.1/mes"
 xmlns:eis="http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
 shortName="ResourceAdapter">

<Property>CEI</Property>
<ElementKind name="Polling">
 <EventNature name="STARTED" eventName="eis:WBI.JCAAdapter.Polling.STARTED" />
</ElementKind>
</EventNaturesSpec>
```

2. ".xsd" file changes

   The .xsd event schema file provides monitoring that is specific to each data element, and it also defines the types of events that can be emitted for the data elements. The following is an example of xsd event schema content::

```
<?xml version="1.0" encoding="UTF-8"?>"
<EventSpec xmlns="http://www.ibm.com/xmlns/prod/websphere/monitoring/6.0.0/es"
 name="Events"
```

```
    targetNamespace="http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
 xmlns:er="http://www.ibm.com/xmlns/prod/websphere/recovery/6.0.0/es/eventpayloads"
>
<Event name="ResourceAdapter.Polling.STARTED" situationType=
"STATUS" situationCategory="ReportSituation"
reasoningScope="EXTERNAL" parent="WBI.MonitoringEvent">
 <Payload>
  <Data name="PollFrequency" type="int" minOccurs="0" maxOccurs="1"/>
  <Data name="PollQuantity" type="int" minOccurs="0" maxOccurs="1"/>
 </Payload>
</Event>
```

3. Invoke Events

```
import com.ibm.j2ca.extension.logging.internal.cbe.EngineData;
  ...
CBEEngineData engineData = CBEEngineDataFactory.getEngineDataForEventType("Polling");
//This will instantiate the EngineData class for user defined event e.g "Polling".
engineData.setValue("EventAction","STARTED");
//This will set the user defined action e.g. "Started"
engineData.setValue("PollFrequency", activationSpec.getPollPeriod());
//This will set the user defined arg
engineData.setValue("PollQuantity", activationSpec.getPollQuantity());
//This will set the user defined arg
```

Performance monitoring infrastructure (PMI) for resource adapters:

The Performance Monitoring Infrastructure (PMI) is the underlying framework in WebSphere Application Server that gathers performance data from various runtime resources such as adapters.

The purpose of monitoring is to observe the progress of execution of WebSphere Business Integration applications, and the WebSphere Business Integration system itself, and publish the results of this observation. By using Performance Monitoring Infrastructure (PMI), you can observe the progress of adapters running in the server runtime environment and other business integration applications, and publish the results. PMI collects data, such as average response time and total number of requests, from various components in the server, and organizes the data into a *tree structure*. You can observe data through the **Tivoli Performance Viewer**, a graphical monitoring tool that is included with WebSphere Application Server..

You can monitor the performance of the adapters by having PMI collect data at the following points:

-
- InboundEventRetrieval:

  Will monitor performance of retrieving events from the EIS. It enables monitoring of entering, exiting, failing of the EventManager.getEvents() method.
- InboundEventDelivery:

  Will monitor the performance when resource adapter deliver data to the endpoint, which conveys changes in or general information from the EIS. It enables monitoring of entering, exiting, failing of the EventSender.doSendEvent() method.
- Outbound:

  Will monitor the performance of outbound processing of a resource adapter. It enables monitoring of entering, exiting, failing of the WBIInteraction.execute() method.

Extending PMI on WebSphere Process Server:

To add a user-defined element or method into the list of monitorable components you need to modify code and schema files.

Purpose

1. Monitorable Element Schema (.mes) file changes

   Defines the element type within an adapter where monitoring can be attached.

   The element type is specified using the Qname of the element type from the schema, which defines the structure of the artifact itself. It also defines the natures (ENTRY, EXIT, FAILURE) that are available for that type of element. The sample below declares how the monitors can be attached to myOutbound. For example myOutbound method can emit event at ENTRY, EXIT or FAILURE event points.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <EventNaturesSpec
    name="EventNatures"
    targetNamespace=
     "http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
    xmlns=
     "http://www.ibm.com/xmlns/prod/websphere/monitoring/6.0.0/mes"
    shortName="JCAAdapter">

     <Property>CEI</Property>
    <ElementKind name="myOutbound">
     <EventNature name="ENTRY" eventName="eis:WBI.JCAAdapter.myOutbound.ENTRY" />
     <EventNature name="EXIT" eventName="eis:WBI.JCAAdapter.myOutbound.EXIT" />
     <EventNature name="FAILURE" eventName="eis:WBI.JCAAdapter.myOutbound.FAILURE" />
    </ElementKind>
   ```

2. ".xsd" file changes

   The xsd event schema file provides monitoring specific of each data elements and it defines the types of events, payload or extended element for each event type that can be emitted for the data elements. Content of schema looks like following:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <EventSpec xmlns=
    "http://www.ibm.com/xmlns/prod/websphere/monitoring/6.0.0/es"
    name="Events"
       targetNamespace=
    "http://www.ibm.com/xmlns/prod/websphere/scdl/eis/6.0.0:JCAAdapter"
    xmlns:er=
    "http://www.ibm.com/xmlns/prod/websphere/recovery/6.0.0/es/eventpayloads"
   >
   <complexType name="WBI.JCAAdapter.myOutbound.ENTRY">
     <complexContent>
      <extension base="wbi:WBIMonitoringEvent" />
     </complexContent>
    </complexType>

    <complexType name="WBI.JCAAdapter.myOutbound.EXIT">
     <complexContent>
      <extension base="wbi:WBIMonitoringEvent" />
     </complexContent>
    </complexType>

    <complexType name="WBI.JCAAdapter.myOutbound.FAILURE">
       <complexContent>
      <extension base="wbi:WBIMonitoringEvent">
       <sequence>
        <element name="FailureReason" type="string" />
       </sequence>
      </extension>
     </complexContent>
    </complexType>

   </schema>
   ```

3. Invoke PMI:

   To invoke PMI statistics around a method named myOutbound, you would do the following:

   a. Import com.ibm.j2ca.extension.monitoring.CEI.EventPoint;

   b. Define a unique PMI event point name.

      For example String eventName = uniqueAdapterID + "##" + ″myOutbound″;

   c. Get an instance of EventPoint:

      for each eventAction ENTRY, EXIT, FAILURE. EventPoint ep = (EventPoint)(EventPoints.INSTANCE.getEventPoints(eventName,eventAction))

d. If eventPoint is enabled, then fire event for Entry, Exit and Failure is invoked.

   Entry event is fired in the beginning of the method call, exit event is fired in the end of the method call and failure event is fired in case of exception. For example we can invoke failure event by following API call.

```
if(ep.isEnabled()) {
          ep.fire(new String[]{"FailureReason"},
          new Object[]{ex.toString()});
}
```

Extending PMI on WebSphere Application Server:

Extending PMI on WebSphere Application Server does not require you to add content into schema files (.xsd and .mes files).

Invoke PMI

Invoking PMI statistics on WebSphere Application Server can be done by following the steps listed below.

Note: The steps below describe how to invoke PMI statistics around a method named myOutbound.

1. Import com.ibm.j2ca.extension.monitoring.CEI.EventPoint;
2. Define a unique PMI event point name.

   For example String `eventName = uniqueAdapterID + "##" + "myOutbound";`
3. 3. Get an instance of EventPoint: for each each eventAction ENTRY, EXIT, FAILURE.

```
EventPoint ep =
(EventPoint)(EventPoints.INSTANCE.getEventPoints(eventName,eventAction))
```
4. 4. If eventPoint is enabled fire event for Entry, Exit and Failure.

   Entry event is fired in the beginning of the method call, exit event is fired in the end of the method call and failure event is fired in case of exception. For example we can invoke failure event by following API call:

```
if(ep.isEnabled()) {
         ep.fire(new String[]{"FailureReason"}, new Object[]{ex.toString()});
}
```

Application response measurement (ARM):

Application response measurement (ARM), an API jointly developed by an industry partnership, monitors the availability and performance of applications. ARM is an approved standard of *The Open Group*.

To ensure that requests are performing as expected in a multi-tiered heterogeneous server environment, you must be able to identify requests based on business importance. In addition, you must be able to track the performance of those requests across server and subsystem boundaries, and manage the underlying physical and network resources used to achieve specified performance goals.

You can collect this performance data by using versions of middleware that have been instrumented with the Application Response Measurement (ARM) standard.

Combining ARM calls within your application with an ARM agent, users of your application will be able to answer questions like the following:

- Is a transaction (and the application) hung, or are transactions failing?
- What is the response time?
- Are service level commitments being met?
- Who uses the application and how many of each transaction are used?

The resource adapters are instrumented with the Application Response Measurement API, an API that allows adapters to collect and manage transaction end-to-end response time and volumetric information.

The adapters can participate in IBM Tivoli Monitoring for Transaction Performance, by allowing collection and review of data concerning transaction metrics.

The resource adapters using ARM define transactions at following three points:

- InboundEventRetrieval:

  Will measure response time of retrieving events from the EIS. It measures response time of the EventManager.getEvents() method.
- InboundEventDelivery:

  Will measure response time when resource adapter deliver data to the endpoint, which conveys changes in or general information from the EIS. It measures the response time of the EventSender.doSendEvent() method.
- Outbound:

  Will measure the response time of outbound processing of a resource adapter. It measures the response time of the WBIInteraction.execute() method.

An ARM agent, such as Tivoli Composite Application Manager for Response Time Tracking, can perform response time collection and analysis.

To enable/extend ARM, different brokers need to implement: armTransactionFactoryName() method found in com.ibm.j2c.monitoring.ARM AdapterArmTransactionFactory class. This transaction factory creates all objects that defined in the org.opengroup.arm40.transaction package.

ARM interface

```
/**
* ArmTransactionFactory provides methods to create instances of the classes in the org.opengroup.arm40.transaction package.
**/
package com.ibm.j2ca.extension.monitoring.ARM;

public interface AdapterARMTransactionFactory {
 public String armTransactionFactoryName();
}
```

Extending application response measurement (ARM) events using the InteractionMetrics:

The WebSphere container provides `InteractionMetrics` interface that introduces the capability for any resource adapter to participate in reporting its usage time in a request and have that time reported by the various request metrics reporting tools available for WebSphere.

Tracking interaction metrics

The WebSphere `ConnectionEventListener` has implemented this class.

WebSphere keeps an `EventListener` associated with each `ManagedConnection` to track the interaction time on a per `ManagedConnection` basis to gather usage time

statistics and data for each `ManagedConnection`, which can be used to assess and troubleshoot performance related problems.

In order for resource adapters to participate in various WebSphere RequestMetric tools for outbound, diagnostic tools, etc, you will need to follow these steps:

1. Import com.ibm.websphere.j2c.*;
2. Get interaction metrics listener by using WBIManagedConnection classes' getInteractionListener() method or by calling WBIInteraction classes' getInteractionListener() method.
3. At the beginning of each method to report statics for, call isInteractionMetricsEnabled, check whether the listener is enabled. listener.isInteractionMetricsEnabled(); If it returns false, do nothing for the rest of this request.
4. If isInteractionMetricsEnabled returns true, call preInteraction in the beginning of a method where you want to start ARM Object ctx = listener.preInteraction();
5. Before sending the request to the downstream EIS process, call getCorrelator and attach the correlator with the request so that the downstream EIS process can get the correlator. Obtain correlation byte[] armCorBytes = listener.getCorrelator();
6. In the end of method you need to collect the ARM statistics by calling postInteraction(ctx, InteractionMetrics.RM_ARM_GOOD, ispec);
7. In case of exception call postInteraction(ctx, InteractionMetrics.RM_ARM_FAILED, ispec);

For details about `interactionMetrics` API please see: l

Extending ARM events using the Open Group API:

Use following reference information to make you component ARM-enabled using the Open Group API.

Application response measurement (ARM) documentation

Use following links to access the information on how to make you component ARM enabled using the Open Group API.:

- Application Response Measurement (ARM) Instrumentation Guide.
- ARM 4.0 Java APIs.
- ARM fundamentals and downloads.

First failure data capture (FFDC):

First failure data capture (FFDC) provides the instrumentation for exception handlers (catch blocks) to record exceptions that are thrown by a component.

To provide FFDC for your component, exception handlers can be *instrumented* by defining an aspect, which determines the packages, classes, methods, and exceptions types that will be supported by FFDC.

Users extend the abstract `FFDCSupport` aspect that captures the required context, such as method name and exception object, automatically. No additional configuration is required because the data provided by the aspect is the same data that would be provided from a hand-coded invocation.

FFDC processing overview

Instead of explicitly instrumenting catch blocks by calling FFDC directly, either manually or by using a tool, you can write a simple aspect using the AspectJ language, which encapsulates the FFDC policy for your code.

The `FFDCSupport` aspect is abstract. Like an abstract class, `FFDCSupport` cannot be instantiated and defers some of its implementation to a sub-aspect. It follows the standard library aspect pattern of declaring an abstract pointcut for which you must declare a concrete implementation. This concrete implementation of the pointcut can use the simple AspectJ scoping pointcut designators (PCD) such as `within()` and `withincode()` to determine the packages, classes and methods to be included in the FFDC policy.

FFDC programming examples

The following examples assume a certain familiarity with the AspectJ language (see http://eclipse.org/aspectj/). In most cases a user of the FFDCSupport aspect will require knowledge of only a small subset of the AspectJ syntax. In particular they should know how to define a concrete aspect by extending an abstract one and how to declare a concrete pointcut typically using simple scoping primitive pointcuts.

```
import com.ibm.websphere.ffdc.FFDCSupport;

public aspect Example_1 extends FFDCSupport {

 protected pointcut ffdcScope () :
  within(com.foo.*);
}
```

Figure 7. Add FFDC to the com.foo package

Figure 1 illustrates the simple aspect `Example_1` that adds FFDC to all classes in the com.foo package. The example illustrates the following processing:

1. On line 1, the `FFDCSupport` aspect is imported
2. On line 3, the `FFDCSupport` aspect is extended and made concrete in a similar way to a Java class.
3. On line 5, the inherited abstract pointcut `ffdcScope()\` is made concrete.

   This is done using the `within()` pointcut designator (PCD) and “*” wildcard that results in FFDC for all classes in the com.foo package. For example, com.foo.Bar.

```
import com.ibm.websphere.ffdc.FFDCSupport;

public aspect Example_2 extends FFDCSupport {

 protected pointcut ffdcScope () :
  within(com.foo..*);
}
```

Figure 2 illustrates aspect `Example_2`, which differs from `Example_1` . Notice line 13, where the wildcard includes ″double dots″ (..) in the within() PCD, which means the includes all classes in the com.foo package and sub-packages. For example, com.foo.impl.Bar.

Figure 8. Add FFDC to com.foo package and all sub-packages

```
import com.ibm.websphere.ffdc.FFDCSupport;

public aspect Example_3 extends FFDCSupport {

 protected pointcut ffdcScope () :
  within(com.foo.*)
  && !within(com.foo.Goo);

}
```

In Figure 3 aspect `Example_3` has the same effect as `Example_1` except it excludes FFDC for class com.foo.Goo by using the && and ! operators to form a *pointcut expression*.

Figure 9. Add FFDC to all classes in the com.foo package excluding com.foo.Goo

```
import com.ibm.websphere.ffdc.FFDCSupport;

public aspect Example_3 extends FFDCSupport {

 protected pointcut ffdcScope () :
  within(com.foo.*)
  && !withincode(* com.foo.Goo.aMethod(..));

}
```

In Figure 4, aspect `Example_4` is also similar to `Example_1`, however FFDC is excluded from a particular method on line 30 using the withincode() PCD.

Figure 10. Add FFDC to the com.foo package but exclude aMethod

```
import com.ibm.websphere.ffdc.FFDCSupport;

public aspect Example_4 extends FFDCSupport {

 protected pointcut ffdcScope () :
  within(com.foo.*)
  && !args(ClassNotFoundException);

}
```

In Figure 5 aspect `Example_5` illustrates how to account for a *programming by exception*, where certain exceptions are not considered to be a failure and should not be reported. In the example, the handling of `ClassNotFoundException` will not be reported to FFDC. The `args()` PCD on line 39 selects a join points based on contextual information in this case the exception passed to the handler join point.

Figure 11. Add FFDC to the com.foo package but exclude catch blocks for ClassNotFoundException handling

If you use the FFDCSupport aspect, you can ensure a consistent FFDC policy for your application by adding *declare warning* or *error advice* to your aspects While this capability is not explicitly provided by the FFDCSupport aspect, you can leverage the use of AspectJ and follow a standard pattern of enforcing a policy implemented using an aspect with compiler warnings or errors.

In Figure 6, aspect `Example_6` illustrates how to prevent direct use of the FFDC API or undesired dumping of exception messages and stack traces to the console. The declare warning statements on lines 51 and 57 instruct the AspectJ compiler to issue warnings during weaving if join points are matched by the accompanying pointcuts on lines 48 and 54 respectively.

The statements are only evaluated during compilation and have no impact on the runtime.

```
import com.ibm.websphere.ffdc.FFDCSupport;

public aspect Example_6 extends FFDCSupport {

 protected pointcut ffdcScope () : within(com.foo.*);

 public pointcut ffdcCall () : call(* com.ibm.websphere.ffdc..*(..));

 declare warning : ffdcCall() && ffdcScope() :
  "Don't call FFDC directly. Use FFDCSupport aspect.";

 public pointcut dumpException () : call(void Throwable.printStackTrace(..));

 declare warning : dumpException() && ffdcScope() :
  "Don't dump exceptions to console. Use FFDCSupport aspect.";

}
```

Figure 12. Warn user about calling FFDC directly or dumping stack traces

When using FFDCSupport aspect you can control the data gathered. Two template methods `getSourceId` and `getProbeId` are provided to you for this purpose. For example, you may want to limit the length of the source ID strings. In Figure 7, aspect Example_7 illustrates how to override the `getSourceId` method and return a short name.

```
import com.ibm.websphere.ffdc.FFDCSupport;

import org.aspectj.lang.JoinPoint;

public aspect Example_7 extends FFDCSupport {

    protected pointcut ffdcScope () :
        within(com.foo.*);

    protected String getSourceId (JoinPoint.StaticPart ejp) {
        String name = ejp.getSignature().getName();
        return name;
    }

}
```

Figure 13. Override default source ID generation to create short name

You can use FFDC to control how your classes are introspected by implementing the `introspectSelf` method. Class Person in Figure 8 illustrates how you can hide a sensitive field (in this example a password is hidden).

```
private class Person {
    private String userid = "USER";
    private String password = "PASSW0RD";

    public String[] introspectSelf () {
        String[] self = { "userid=" + userid, "password=XXXXXXXX" };
            return self;
        }
}
Hide password field of Person class from introspection.
```

Figure 14. How to hide a sensitive field in this case a password

Exception messages

Exception messages, like trace messages, convey information about problems. The difference is that exception messages are tailored more directly to support teams familiar with adapter source code, and therefore need not be translated.

Treat text included in exception messages as you would text for trace messages. In general, exception messages are not directed at general users but rather at support teams who have the ability to investigate adapter source code. For that reason, exception messages need not be translated and can be hard-coded. If an exception is thrown and the customer must be informed of the problem, use an appropriate log message, which can be translated, to inform the customer; refrain from simply printing out the contents of the exception.

When implementing an adapter exception message, do the following:

- Define subclasses of `ResourceException` where appropriate and include relevant properties on these implementations
- Raise the exception with a message that can be hard-coded in English.
- Print the stack trace of the exception using the trace API. This allows support teams to see all exception details.
- When appropriate, the exception messages raised should have a corresponding high level message in the `LogMessages.properties` file. The developer should log this high-level message after raising the exception so that general users can see an explanation of the error in their native language.

Changing the Java logging API settings

To change the Java logging API settings, you modify a permission in the adapter deployment descriptor.

The following permission must be added to the adapter deployment descriptor, represented as the file `ra.xml` in the RAR package:

```
permission java.util.logging.LoggingPermission "control";
```

In the `ra.xml` file, this permission is as follows:

```
<security-permission>
  <security-permission-spec>
    grant {
  permission java.util.logging.LoggingPermission "control";
     };
  <security-permission-spec>
<security-permission>
```

Validating the code

Validate your adapter implementation by unit testing it outside of a JCA container (unmanaged testing mode). You can then deploy to the target runtime server and test instances of the adapter (managed testing mode).

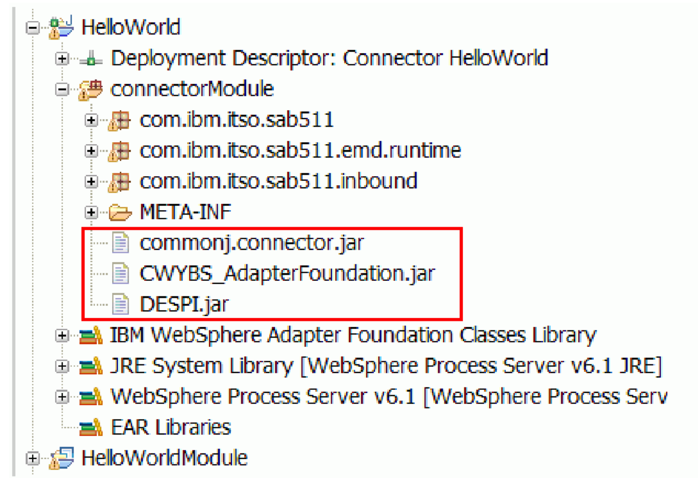
Unmanaged and managed testing modes are not mutually exclusive. A thorough testing regime typically involves unit testing to debug and refine adapter components prior to managed testing in the target environment.

Testing enterprise metadata discovery (EMD) of the adapter

Testing the EMD implementation means testing if the adapter can connect to EIS and discover services from an existing metadata repository or is able to build the appropriate interactions with the EIS by generating the required artifacts.

To test the enterprise metadata discovery (EMD) implementation for the developed resource adapter, complete the following steps:

1. From the IBM WebSphere Adapters Foundation Classes library, copy the following three dependent jars into the **connectorModule** connector project:
   - commonj.connector.jar
   - CWYBS_AdapterFoundation.jar
   - DESPI.jar

   Displaying the dependent jars

   Note: Alternatively, if you want to export the jars as RAR, you can use the **AFCUtility** tool to place these three jars into the RAR. This tool is available as WAT plug-in from `/plugins/com.ibm.j2ca.wat.afc_6.2.0/AFCUtility`.
2. In the WebSphere Integration Developer window, click **Go to the Business Integration perspective**.
3. Right-click inside the Business Integration section of the WebSphere Integration Developer window.
4. Type in a new **Module Name** in the New Module window. Click **Finish**.
5. Open the external service by clicking the **File->New->External Service**.
6. In the External service window, expand Adapters and select the adapter name entered in the ra.xml file.

   Note: You can enter the adapter name in the ra.xml file using the Resource Adapter Deployment Descriptor editor.
7. Click through **Next** to open the last External service window. Click **Browse** and select the module created in Step 4 to generate the required artifacts.

   Note: If you do not add the dependency jars to the connectorModule, the following error might display during the deployment:
   `java.lang.NoClassDefFoundError: com.ibm.j2ca.base.WBIResourceAdapter`

Testing the adapter in unmanaged mode

Testing in unmanaged mode means unit testing the adapter implementation in your development environment. You can unit test your adapter with JUnit, a widely used and reliable open source framework for regression testing.

JUnit: an open source framework for unit testing

JUnit is becoming the standard tool for unit testing in Java development environments. JUnit allows you to quickly and easily integrate automated regression testing into your coding and build processes.

With JUnit, an open source unit test framework, you can write and run tests for your adapter implementation. You use a simple `setup()` method to prepare each component for testing. Each test method can contain one or more assertions. The assertions test actual results against expected results. Using JUnit assertions, you can achieve a high degree of code quality and responsiveness to requirements outside of the adapter runtime environment.

A simple JUnit test case resembles the following:

```
public class MyTest extends TestCase {
    protected void setUp() throws Exception {
        super.setUp();
    }
    public testSomething() throws Exception{
 String result = classUnderTest.executeSomething();
 assertEquals(result,"Something");
     }
}
```

The `setUp()` method is called once before each test method. The purpose of the `setUp()` method is to prepare the component for the test. You can create several test methods; each must begin with the word `test` and contain at least one assertion.

The next section shows how to use the `setUp()` method to prepare your adapter for testing, and how to use the test methods to execute functions. Because the goal is to test the adapter in unmanaged mode, you must run the adapter outside of a JCA container.

Validating the adapter in unmanaged mode–testing the adapter as part of development, gradually building the test case suite to address requirements–is a first step. This prepares the adapter for managed testing in a runtime environment. It also yields useful artifacts: the test case suite remains useful after unit testing because changes in code that break the functionality are tracked, alerting you when testing future iterations inside the development environment.

For more information about JUnit, see http://junit.org.

Developing JUnit tests

You unit test outbound and inbound processing by creating and specifying JCA contracts and operations. You then test and compare data before and after applying the tests. The TwineBall sample helps illustrate these steps.

Outbound

The J2EE Connector Architecture (JCA) specification defines an unmanaged mode for running adapters. This means running the adapter outside of a JCA container and in process with the caller. This is the environment for developing JUnit tests.

Through a series of common client interface (CCI) and service provider interface (SPI) calls, you can force the adapter to perform an operation you want to test. First you must create instances of `ManagedConnectionFactory` and `ResourceAdapter` and then set the appropriate properties in the client code.

Your adapter may or may not be dependant on "live" data inside the EIS. If so, you must either return the data to a known state after every test, or create a mock implementation of the EIS API so that EIS data remains untouched.

setUp()

In the setup method for outbound, perform the following step:

1. Load any schemas (if necessary)

Test

When you have completed the steps for setting up the test, you are ready to call the CCI interaction and validate the result. The procedure below creates an object, retrieves it, and validates its content.

In the test method, you will need to do the following things:

Note: These tasks can be delegated to helper methods

1. Create the data and metadata you will be working with (JavaBean, SDO, or other format).
2. Create a DataExchangeFactory for this data type, set the data as the bound object
3. Create a StructuredRecord and initialize it, with the Data Exchange Factory and the metadata
4. Invoke the adapter's interaction.execute() using JCA CCI semantics
5. Inspect the result of the output, by getting the bound object of the data exchange factory of the output record.

1. Create a new WBIRecord.
2. Create a business object, populate it with data, and place it in WBIRecord.
3. Set the appropriate verb in the business object.
4. Call the interaction, capturing the output.
5. Perform the assertions, which include the following:
   - Retrieving the object
   - Validating the retrieved object against the original data

For a detailed and coded example of this procedure, see `TwineBallInteractionTest`.

Inbound

In contrast to the outbound direction, inbound communication is not defined for unmanaged operation. To test the adapter's inbound capability outside of a JCA container, you must implement some of the JCA container contracts. These include the following:

- `BootstrapContext` – To obtain a reference to a work manager and a timer
- `InboundListener` and `MessageEndpoint` – For the listener client
- `MessageEndpointFactory` – To create endpoints
- `WorkManager` – To create work instances
- `Work` – To map to threads

For a detailed and coded example of these contracts, see the TwineBall sample.

setUp()

The `setUp()` method for inbound is very similar to that for outbound. You may, however, not need an outbound connection through the adapter. Accordingly, you need only perform the following tasks:
1. Create an adapter instance and set its properties
2. Start the resource adapter.

Test

The test methods for inbound must do the following:
1. Prepare the data inside the EIS. In a polling adapter, this would cause the triggers to fire, populating the event table.
2. Create a `MockEndpointFactory`, and an `ActivationSpecWithXid`.
3. Call `endpointActivation()` on the adapter.
4. Wait for the event to arrive.
5. Assert the correctness of the data after it is sent to the endpoint.

Note: You can group your inbound and outbound tests into a single suite, and run them at once, getting instant feedback in your development environment.

For a detailed and coded example of the procedures described above, see the TwineBall sample that comes with the WebSphere Adapter Toolkit.

Build and execute TwineBall JUnit

To build and execute TwineBalll JUnit, perform the following steps:
1. Import in the Twineball, or KiteString source code from the samples gallery.
2. Find the testcase you want to run. For example, choose AllTest.java in the `com.ibm.j2ca.sample.twineball` package in the tests folder.
3. Right-click on the class and choose **Run As** → **JUnit Test**.

See the results after the JUnit run. The color green indicates that the test case was run successfully.

Testing the adapter in managed mode

Testing the adapter in managed mode means testing adapter instances on WebSphere Process Server. This type of testing, in contrast to testing in unmanaged mode, more closely reflects the production environment that customers encounter.

Before testing your adapter implementation in managed mode, you must export a resource adapter EAR file to WebSphere Process Server.

Running tests in managed mode help uncover problems your adapter might have with the following services provided by WebSphere Process Server and the Adapter Foundation Classes:
- Connection management
- Transaction management
- Event management

For information on testing the adapter in managed mode in WebSphere Application Server, see Validating code with Rational Application Developer / Websphere Application Server.

Installing the test client

To test your adapter in a runtime environment, you must first install a test client on the target WebSphere Process Server.

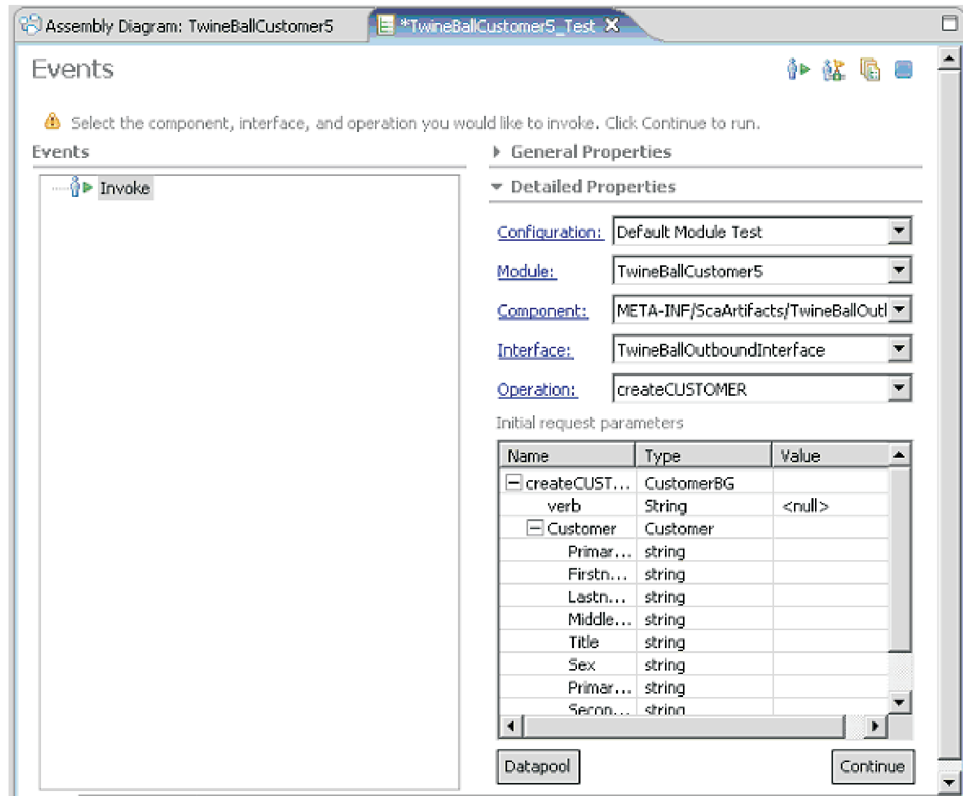
1. Install the test client, `TestController.ear`, on the target WebSphere Process Server. Locate the file `<WebSphere Integration Developer>\eclipse\plugins\com.ibm.wbit.comptest.core\TestController.ear` on your WebSphere Integration Developer system (this file was installed with WebSphere Adapter Toolkit). Follow the steps in "Creating and exporting a resource adapter" on page 206.
2. Apply the CompTest patch to the target WebSphere Process Server if the server is not installed on the same machine as WebSphere Integration Developer. If you installed the test client on a machine that is not running WebSphere Integration Developer, (for example on an AIX, HP-UX or Solaris workstation that is running WebSphere Process Server), then you must install this patch. The patch is located on the WebSphere Process Server ifix website and contains two JAR files: `CompTestCommon.jar` and `CompTestController.jar`. To install them, unzip the patch to the root directory of WebSphere Process Server.

Testing outbound functionality

You test outbound processing by configuring an adapter instance, selecting test parameters, and optionally executing the test in debug mode to pause at breakpoints.

After you have created and exported your adapter to WebSphere Process Server, you can test outbound functionality by following the procedure below.

1. Open the test module in the Assembly Editor. Right-click the test module in the Navigation pane and select **Test** → **Test Module**.
2. Configure the adapter instance. The Test Client displays a panel in which you select the **Configuration**, **Module**, **Component**, **Interface** and **Operation** you want to test. Make these selections, including the verb and the value(s) you want sent to the adapter and click **Continue**.



Test module configuration

3. Select the testing mode and click **Finish** to start the test. On the Deployment Location screen, select a WebSphere Process Server to test in managed mode (optionally you can select **Eclipse** to test in unmanaged mode). In addition, you select **Run** or **Debug** mode. If you select **Debug** mode, you can set breakpoints in your code; when the test reaches a breakpoint, WebSphere Integration Developer displays the Debug perspective.

   A screen informs you that the test is running. Then, if the adapter is successful, the business object you specified is populated with the return data.

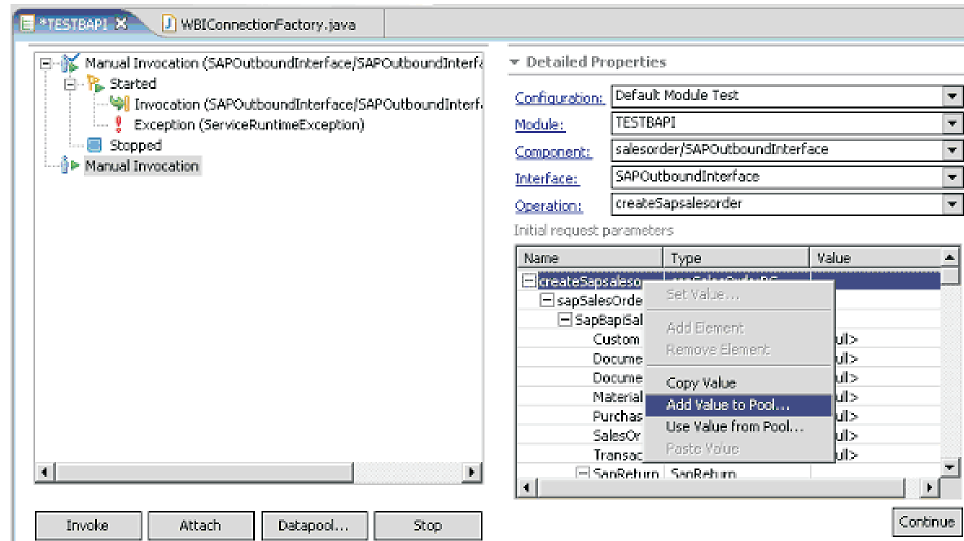
To run another test, click on **Invoke** in the top right corner of the Assembly Editor.

Saving business object data

You can use Datapool to save business object data during testing. This eliminates the need to reenter business data with each test iteration.

After entering the same data numerous times, you will be thankful for Datapool, a WebSphere Integration Developer utility. Datapool allows you to save business object data for subsequent tests. To use Datapool, perform the steps in the following procedure:

1. Open the Test module in the Assembly Editor. Right-click the adapter module in the Navigation pane and select **Test** → **Test Module**.
2. Save the business data. After entering values for Detailed Properties, right-click the topmost entry in the Initial request parameters pane. Select **Add Value to Pool** from the menu.



Adding a value to the Datapool

This adds the data to Datapool.

When you want to use this input data again, select **Use Value from Pool.**

Using an execution trace

The test client you installed provides you with a trace of the execution and the data path of the test. You can optionally load any previously saved execution trace into the test client. This enables you to renew a test session at the point where you saved the execution.

The procedures below describe how to load and save an execution trace file.

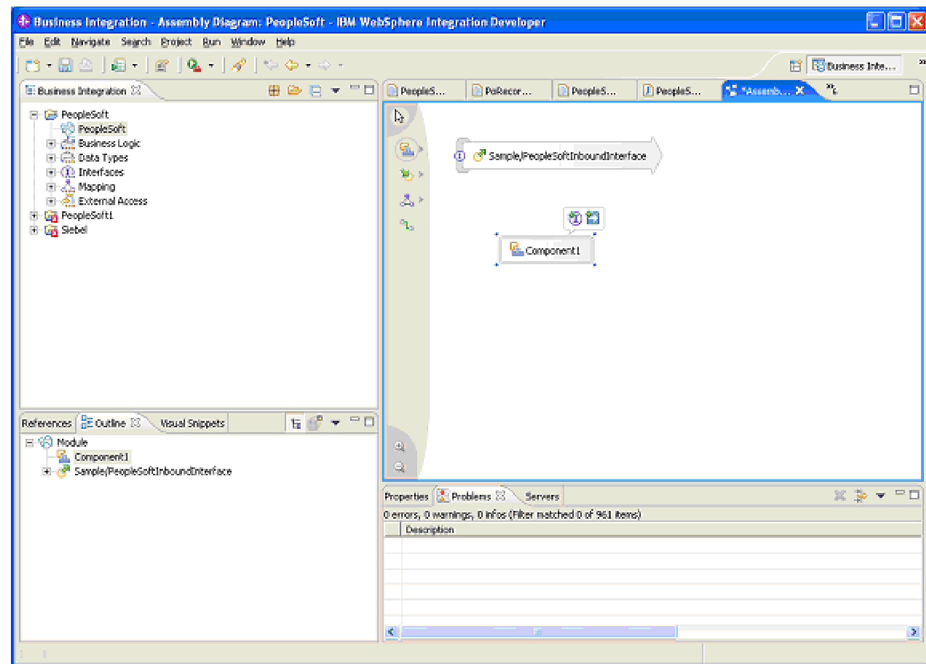
1. Select the module from the Navigation window, right-click and select **Test > Load Execution Trace**.
2. In the **Choose a file** list box, expand the folder that contains the execution trace file that you want to load and select the execution trace file. (By default, execution trace files have a file extension of `.wbiexetrace`.)
3. Click **Finish**. The selected execution trace opens in a new instance of the test client.
4. Save the execution trace.
   a. In the test client, enter **Ctrl-S**.
   b. In the Save dialog, under the **Enter or select the parent folder** field, select the folder where you want to save your execution trace file.
   c. Enter the name that you want to assign to the execution trace file in the **File** name field.
   d. Click **Finish**. The execution trace file is saved to the folder that you selected and the test client page tab changes to the name of the execution trace file.
   e. Enter **Ctrl-S** in the test client to re-save the file.

Testing inbound functionality

To test inbound functionality, you configure an inbound instance of your adapter with an export monitor. You then run an outbound adapter instance to generate an event of interest for your inbound adapter.

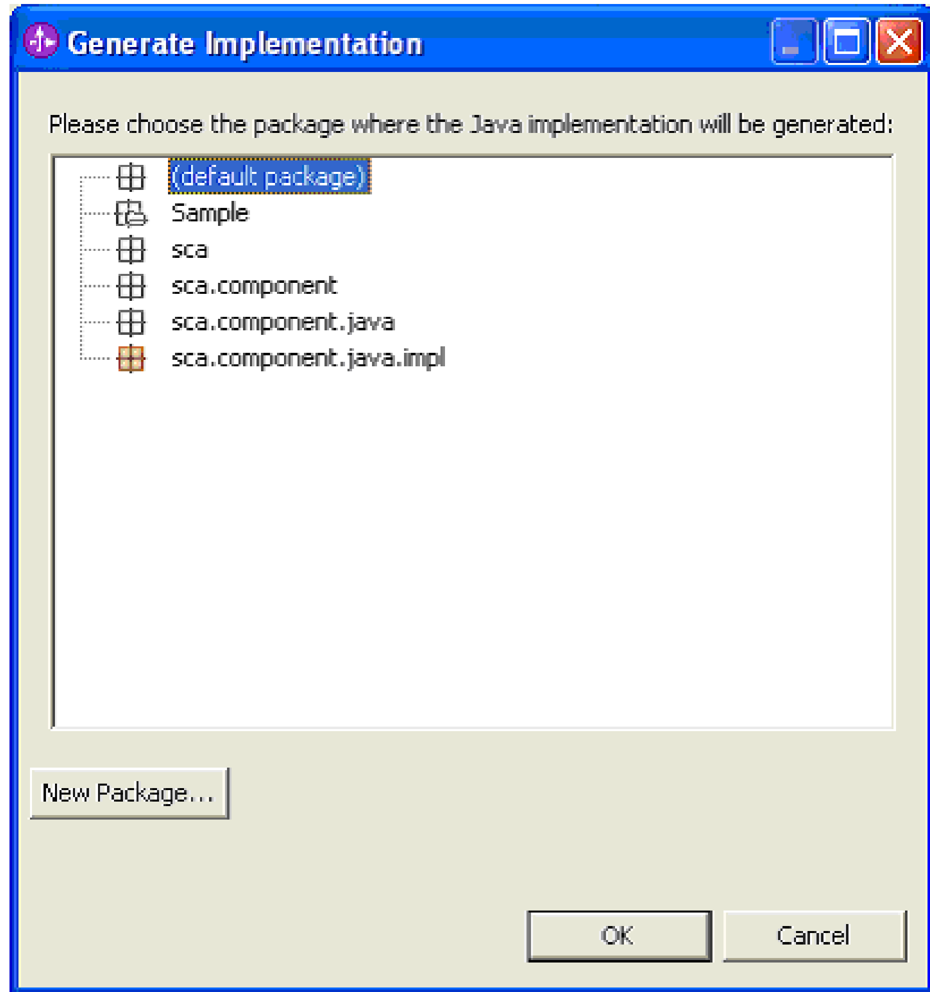
After you have created and exported an adapter EAR file with the service type set to Inbound, you can test inbound functionality.

1. Edit your module using the Assembly Editor, connecting the export to a Java component.
   a. Double-click on the module to start the Assembly Editor.
   b. Create a new Component by selecting the Component with no implementation from the first option.
   c. Add a wire by clicking the export and dragging it to the component.



Wiring a component

   d. Click **OK** in the next window.
   e. Right-click the component and select **Generate Implementation** → **Java** . This creates a Java component that simulates an end point.
   f. Select the package where the Java code should be created. Once the package is selected, the Java file should display as shown in the following figure. This can be edited to insert print or process statements for testing.



Selecting the Java package

g. Save the module.

2. Publish the application to WebSphere Process Server.
3. Open the administration console for the WebSphere Process Server and configure the application’s activation specifications so that it can process inbound requests.
4. Restart the inbound application and ensure that it is polling.
5. To start testing, select the module from the Navigation window, right-click and select **Test** → **Attach** . The test client displays the Events window.
6. Examine the window for an export monitor.
7. Return to the Events tab and click **Continue**. The Deployment Location displays.
8. Select the server on which you want to test and click **Finish** . The Starting The Integration test client window displays.
9. Create an event in your application’s event table. You can do this by running the test client on the Outbound application. When an event is received on the monitored component, an entry will appear in the Events window.

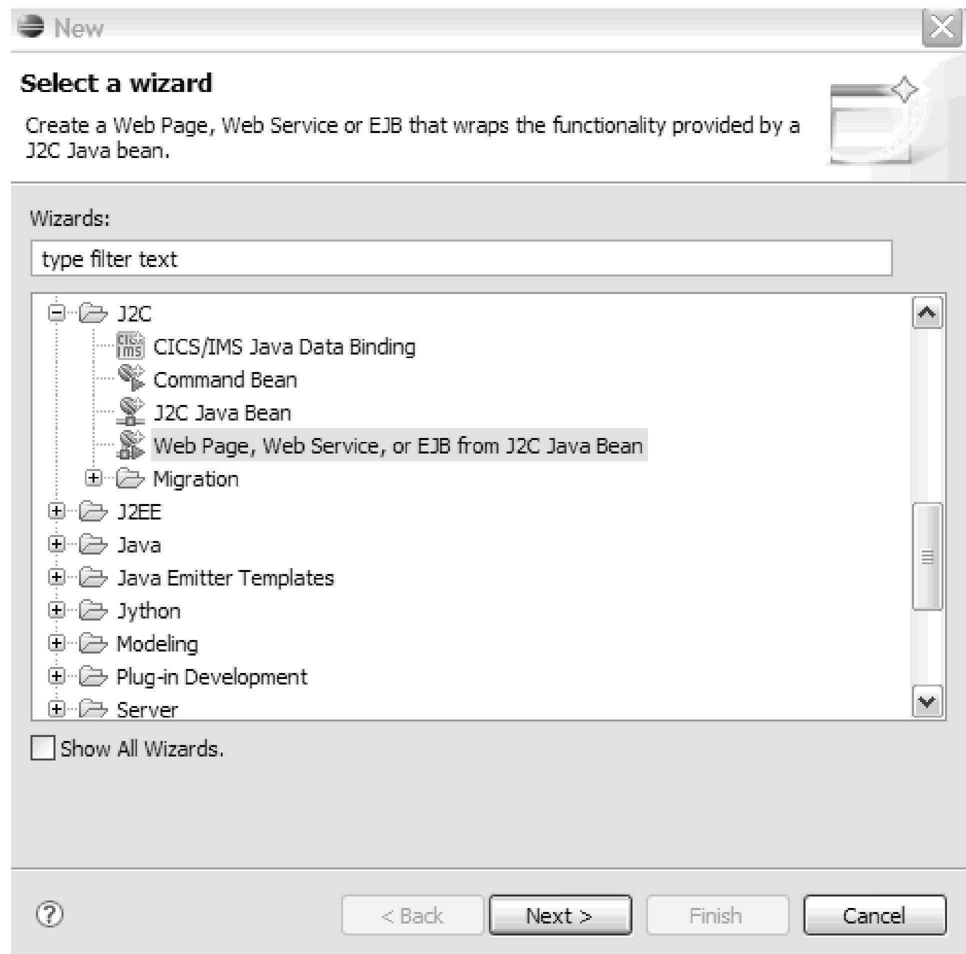
Validating code with Rational Application Developer and WebSphere Application Server

To test the adapter in the Websphere Application Server environment, use the javabean generation capability of EMD to generate records and a java proxy interface to the adapter. Then generate a session bean that will call this interface, and use the Websphere universal test client (UTC) to send data to the adapter.

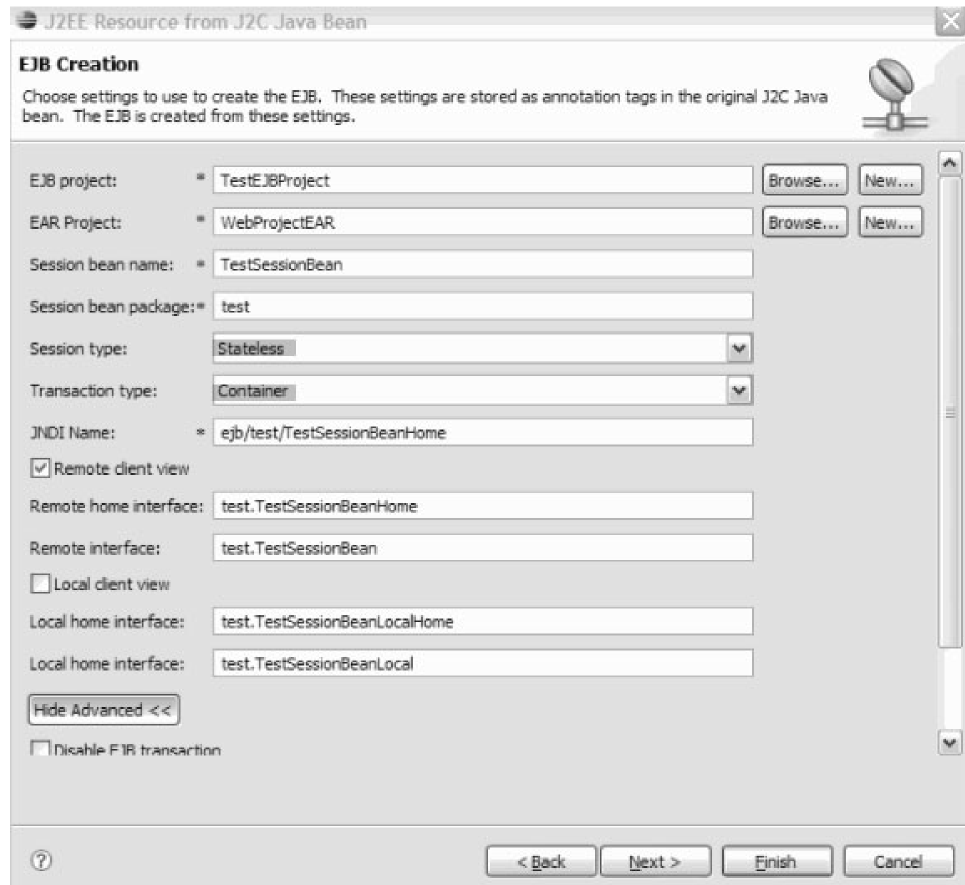
1. Run EMD In Rational Application Developer, EMD can be accessed via **New → Other → J2C JavaBean**

   The output of EMD will be JavaBeans for your schema definition, as well as a "J2C JavaBean" that proxies the adapter.

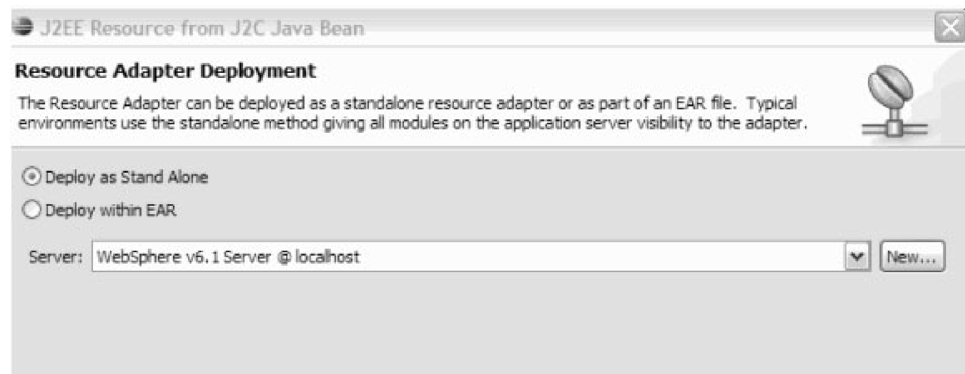
   Once the J2C Beans are generated you need to generate EJBs to Test them.
2. Select the J2C JavaBean that you just created.

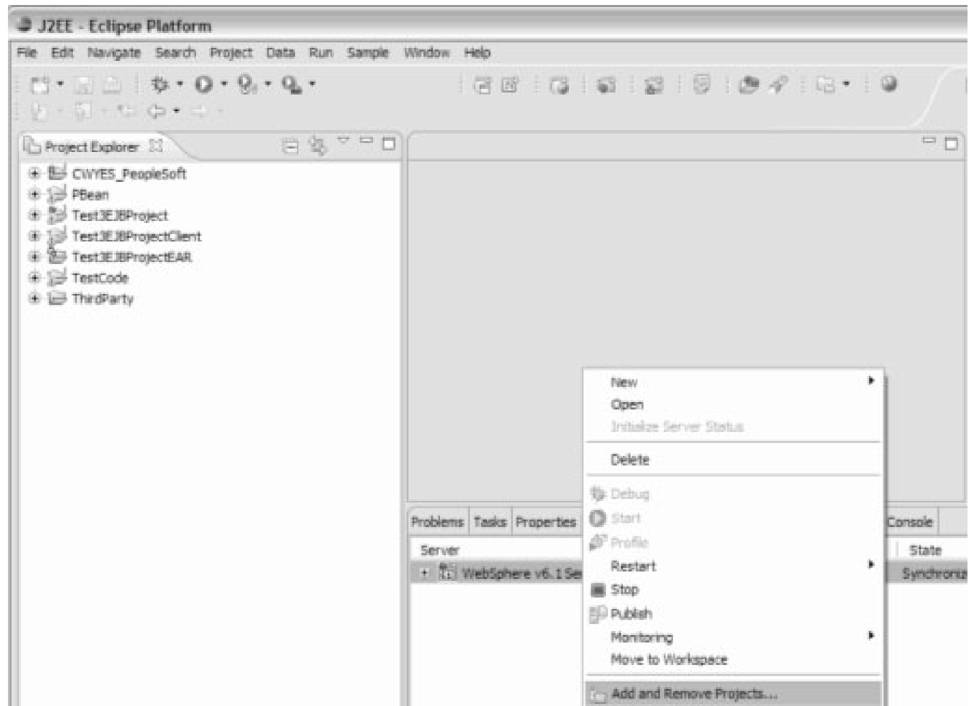
3. In the EJB Creation window, select a **stateless** as the Session type and **container** as the transaction type and click **Next**



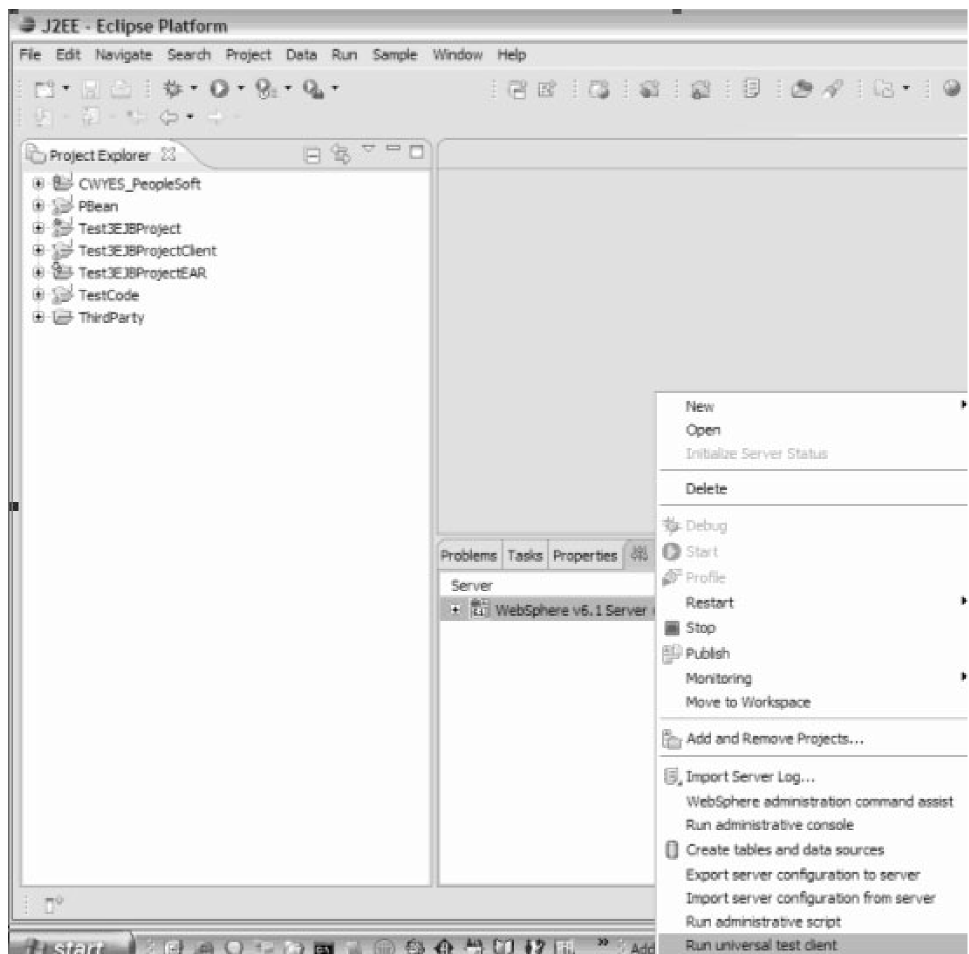
4. In the Resource Adapter deployment panel, choose how to deploy the adapter. You can deploy the adapter with the EAR or you can deploy the adapter as a stand-alone component.



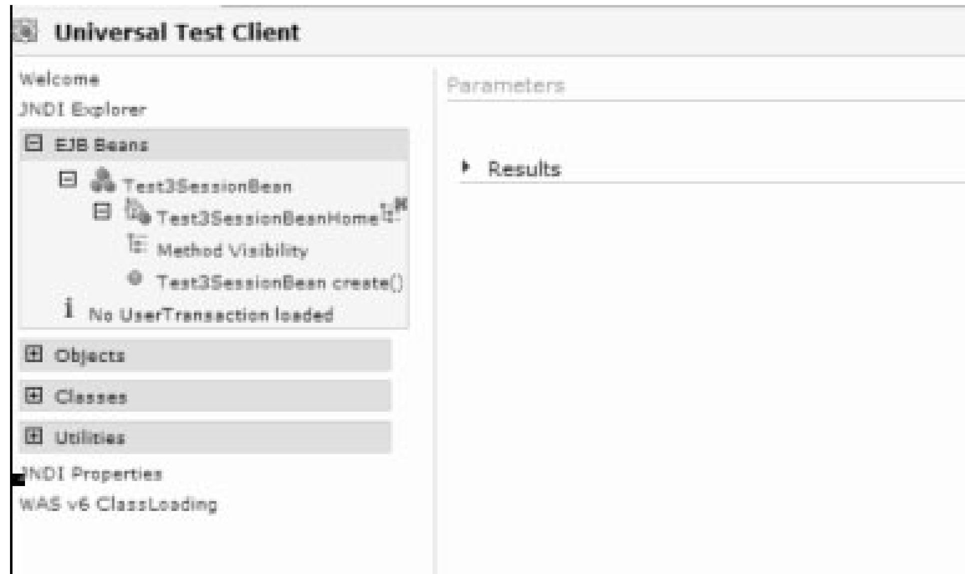
5. Click **Finish** to generate the code. Once the EJB is generated you can send data to the adapter and examine the return values using the UTC.
6. Use the UTC to validate that the adapter can process data Publish your EAR project to the server using the **Add and Remove projects** option.



7. Start UTC using the **Run universal test client** option.



8. Once the UTC comes up, use the JNDI explorer to find your EJB. Look for your session EJB under EJB Beans.



Now, you can test your adapter via the EJB interface.

You can create a session bean using the home interface (create), then invoke business methods on the remote interface, providing the appropriate data. This works the same way as testing any session bean.

Creating and exporting a resource adapter

For every adapter implementation in resource adapter archive (RAR) format, you must create (export) one or more enterprise archive (EAR) files, which you deploy to WebSphere Process Server.

Please note these guidelines when using the procedures below:

- Step 1 describes how to create an EAR file from an adapter RAR file.
- Step 2 describes how to add WebSphere Process Server to WebSphere Integration Developer. If the target server–the server on which you will install the EAR file–is not listed in the Servers window of WebSphere Integration Developer, you must add it.
- Steps 3 through 6 describe how to export or deploy an EAR file to WebSphere Process Server.

1. Create an enterprise archive (EAR) file and export (also known as deploy) it to WebSphere Process Server. Here is a summary of the steps needed to create and export an enterprise application archive (EAR) file for your custom adapter. These steps are defined in adapter-specific detail in IBM WebSphere Adapter user guide documents.
   a. Launch IBM WebSphere Integration Developer.
   b. Create a project in WebSphere Integration Developer and then import the adapter RAR file.
   c. Switch to the **Business Integration** perspective.
   d. Create a module (for example, TwineballCustomerOutbound).
   e. Right-click on the frame and select **New** → **External Service**.

Note: *External service discovery* is equivalent to *enterprise metadata discovery*.

f. Select the appropriate external service.

g. Specify the connection properties, metadata, and service functions.

h. On the **Saving Properties** pane, save the properties to the module you created in Step d.

i. Save the file.

2. Add the target server to WebSphere Integration Developer.

Note: If you have not done so, you must add the target server–WebSphere Process Server–to WebSphere Integration Developer. You must do this before you can export your adapter EAR file for runtime testing.

a. Right-click in the Servers window and select **New** → **Server** from the menu.

b. On the next panel, specify the **hostname** Note that the workstation can run any operating system (Windows, Solaris, AIX, and so on) that supports WebSphere Process Server.

c. Click **Next** to display the WebSphere Server Settings panel. You can also click **Detect** to verify that WebSphere Process Server is installed on the machine.

d. (Optional) Start WebSphere Process Server. Right-click the target WebSphere Process Server in the Servers window and select **Start** from the menu.

e. (Optional) Start the administrative console. Right-click the target WebSphere Process Server in the Servers window and select **Run administrative console** from the menu.

3. Export and deploy an EAR file.

a. In the J2EE Perspective, right-click the adapter EAR file.

b. Select **Export** → **EAR** file from the pop-up menu.

c. When prompted, specify the filename where the EAR is to be saved.

4. Start the target IBM WebSphere Process Server.

5. Start the administration console on that server.

6. Install the adapter EAR file on the target WebSphere Process Server.

a. After the administrative console completes its startup on the target server, log in and navigate to **Applications** → **Enterprise Applications**. Click **Install**.

b. Select the EAR file that you exported previously. Follow the screens, clicking **Next** and eventually**Finish**. Ensure that the EAR has been deployed successfully and save the configuration by clicking on the **Save to Master Configuration** link and following the prompts.

c. Start the application.

Select the application and click **Start**. The status should change from a red X to a green arrow.

d. Ensure that the types in the WSDL are correct.

Switch to the **Resources** perspective, right-click the WSDL file and open the **Interface Editor**. Examine the values in the **Type** column and ensure that these are correct. Usually, the type is a **BusinessGraph**.

e. Publish the artifacts to WebSphere Process Server. Right-click WebSphere Process Server and click **Publish**.

f. Examine the Console window to ensure that your module has been successfully deployed to the server.

Reference

Terminology

The terminology presented are of terms that are used frequently in the documentation.

Adapter foundation classes (AFC)
Sometimes referred to as base classes, the adapter foundation classes are a common set of services for all IBM WebSphere resource adapters. The Adapter Foundation Classes conform to, and extend, the Java 2 Connector Architecture JCA 1.5 specification. The foundation classes include generic contracts and methods to develop a working resource adapter and are included as a component to the WebSphere Adapter Toolkit.

Application response measurement (ARM)
An application programming interface (API), developed by a group of technology vendors, that can be used to monitor the availability and performance of business transactions within and across diverse applications and systems.

Application-specific information (ASI)
The portion of business object metadata that enables the adapter to interact with its application or a data source.

Common Client Interface (CCI)
The Common Client Interface (CCI) of the J2EE Connector Architecture provides a standard interface that allows developers to communicate with any number of Enterprise Information Systems (EISs) through their specific resource adapters, using a generic programming style. The CCI is closely modeled on the client interface used by Java Database Connectivity (JDBC), and is similar in its idea of Connections and Interactions. The generic CCI classes define the environment in which a J2EE component can send and receive data from an EIS.

Common event infrastructure (CEI)
The implementation of a set of APIs and infrastructure for the creation, transmission, persistence, and distribution of business, system, and network Common Base Events.

Data exchange service provider interface (DESPI)
The interface by which resource adapters and runtime components exchange business object data. It is based on the concept of *cursors* and *accessors*, abstracting the data type so that an adapter can be written once and work on runtimes supporting different data types, such as data objects and JavaBeans. DESPI is an architecture that provides the capability of using JCA adapters on multiple brokers (runtime environments). Version 6.1.x and version 6.2 of the Adapter Foundation Classes support the DESPI architecture.

Deployment descriptor
An XML file that describes how to deploy a module or application by specifying configuration and container options. For example, an EJB deployment descriptor passes information to an EJB container about how to manage and control an enterprise bean. Typically deployed in a runtime environment to discover the configuration attributes of the component being described.

Eclipse
: An open source infrastructure for building tools such as an Integrated Development Environment (IDE). The toolkit's wizard and editor are Eclipse plug-ins.

Eclipse Plug-in
: A module that extends the functionality of the Eclipse Platform

Editor
: A component in Eclipse that allows data to be edited. Editors may perform syntax validation.

Enterprise information system (EIS)
: The applications that comprise an enterprise's existing system for handling company-wide information. An enterprise information system offers a well-defined set of services that are exposed as local or remote interfaces or both.

Event
: An occurrence of significance to a task or system. Events can include completion or failure of an operation, a user action, or the change in state of a process. Events generally result from user-defined triggers set on objects in the EIS

IBM Rational Application Developer
: An IBM application that provides a set of extensions to the base Eclipse platform.

IBM WebSphere Adapter
: A J2C Resource Adapter that is based on the Adapter Foundation Classes.

IBM WebSphere Process Server
: Enables deployment of standards-based integration applications in a service-oriented architecture (SOA).

IBM WebSphere Integration Developer
: An IBM IDE.

Inbound
: Inbound is a description of the direction in which data and messages pass from the EIS to a J2EE client application. Adapters support both inbound and outbound data flow.

JCA contract
: A contract is a collaborative agreement between an application server and an EIS on how to keep all system-level mechanisms, such as transactions, security, and connection management, transparent from the application components.

J2EE J2C Resource Adapter
: A system-level software driver that is used by an EJB container or an application client to connect to an enterprise information system (EIS). A resource adapter plugs in to a container; the application components deployed on the container then use the client API (exposed by adapter) or tool-generated, high-level abstractions to access the underlying EIS.

managed mode
: An environment in which connections are obtained from connection factories that the J2EE server has set up. Such connections are owned by the J2EE server.

Outbound
Outbound is a description of the direction in which data and messages pass from a J2EE client application to the EIS. Adapters support both inbound and outbound data flow.

Performance monitoring infrastructure (PMI)
A set of packages and libraries assigned to gather, deliver, process, and display performance data. PMI is the underlying framework in WebSphere Application Server that gathers performance data from various runtime resources such as adapters. The purpose of monitoring is to observe the progress of execution of WebSphere business integration applications, and the WebSphere business integration system itself, and publish the results of this observation.

Request
In a request and response interaction, the role performed by a business object that instructs an adapter to interact with an application or other programmatic entity.

Subclass
In programming, to add custom processing to an existing function or subroutine by hooking into the routine at a predefined point and adding additional lines of code.

Wizard
A sequence of dialogue pages which collect user input to perform a task such as creating a New Java Project in the workspace or creating a New Java Class within a selected project.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2006, 2008

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department 2Z4A/SOM1
294 Route 100
Somers, NY 10589-0100
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows: (c) (your company name) (year). Portions of

this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning:

Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

This product includes software developed by the Eclipse Project (http://www.eclipse.org).

Index

Special characters

.WBIOutboundServiceDescriptionImpl 39
(CCI), Common Client Interface 4
(EAR) project, Enterprise Application Archive 31
(EAR), enterprise application archive 2
(RAR), resource adapter archive 2

Numerics

1.5 specification, Java 2 Connector Architecture JCA 10
2 Connector Architecture JCA 1.5 specification, Java 10

A

ActivationSpecWithXid 67
adapter archive (RAR), resource 2
adapter classes, inbound 37
adapter classes, inbound JCA 50
adapter classes, outbound 35, 40
adapter classes, outbound JCA 49
Adapter Foundation Classes 4, 9, 10
Adapter Toolkit, WebSphere 1
Adapter, WebSphere 3
after-image Create 74
after-image Create operation 74
after-image Update 75
after-images versus deltas 71
Application Archive (EAR) project, Enterprise 31
application archive (EAR), enterprise 2
Application response measurement (ARM) 187
 interface 188
Application-specific information format
 JavaBean Metadata 116
 SDO to JavaBeanRecord mappings 117
ApplyChanges 74
Architecture JCA 1.5 specification, Java 2 Connector 10
Architecture overview 4
 adapter foundation classes component 6
 Application interface component 7
 common services component 6
 connectivity subcomponent 6
 Data exchange subcomponent 6
 JCA connector component 5
 Metadata component 7
 runtime component model 4, 5, 6, 7
Archive (EAR) project, Enterprise Application 31
archive (EAR), enterprise application 2
archive (RAR), resource adapter 2

C

Callback event sender constructors 92
CCI Record 70
Classes
 Adapter Foundation
 WBIFunctionBuilder 160
 WBIMetadataBuild 160
 WBIMetadataType 160
Classes, Adapter Foundation 4, 10
classes, command pattern 43
classes, data binding 38, 48
classes, enterprise metadata discovery 39, 47
classes, inbound adapter 37
classes, inbound JCA adapter 50
classes, JCA enterprise metadata discovery 50
classes, outbound adapter 35, 40
classes, outbound JCA adapter 49
Client Interface (CCI), Common 4
command pattern classes 43
Common Client Interface (CCI) 4
Config Property 67
Connection 49
Connection Definition Property 67
ConnectionFactory 49
ConnectionMetaData 49
ConnectionMetaDataImpl 35
ConnectionRequestInfo 49, 67
ConnectionSpec 35, 49
Connector Architecture JCA 1.5 specification, Java 2 10
Connector Project 30
Connector Project, Java 9
Create operation, after-image 74
custom operations 79

D

Data and Metadata
 about 115
 Application-specific information 116, 117
 JavaBean 115, 116
 relationship to DESPI 115
 SDO 117
 XSD 115
Data binding
 generator 167
 getBusinessObjectName 167
 getDataObject 166
 getNamespaceURI 167
 getRecord 167
 implementation 166
 implementing 159
 interfaces 166
 methods 166, 167
 setDataObject 166
 setRecord 167
data binding classes 38, 48
Data handler
 implementing 158
data model
 business object 69
DataBindingImpl 38
deployment descriptor 10
 description of 56
 editor 55, 56
descriptor, deployment 10
discovery classes, enterprise metadata 39, 47
discovery classes, JCA enterprise metadata 50
discovery, enterprise metadata 3

E

Eclipse plug-in 9
editing source 67
enterprise application archive (EAR) 2
Enterprise Application Archive (EAR) project 31
enterprise metadata discovery 3
 application adapters 124
 build packages 158
 data bindings 158
 data handlers 158
 implementation 124
 technology adapters 158
 technology-style adapters 157
enterprise metadata discovery classes 39, 47
enterprise metadata discovery classes, JCA 50
event error handling 89
EventStore 37, 89
exception messages 193
Extending PMI on WebSphere Application Server 187
Extending PMI on WebSphere Process Server 186

F

Fault handling 168
 binding 168
 business object utility 168
 configuration 172
 defining custom faults 173
 fault names 172
 implementing 169
 naming faults 169
 selector 168
 support 169
First failure data capture (FFDC) 189
Foundation Classes, Adapter 4, 10
Function selector
 implementing 158

© Copyright IBM Corp. 2006, 2008

G

generation options 35

H

hardware requirements 12

I

implementation overview 68
Inbound
  callback event notification 90, 91, 92, 94
  callback event sender 91
  callback event sender constructors 92
  event notification 79
  one way callback events 90
  operations 72
  request and response callback events 90
  standard 72
  using adapter foundation classes 91
  with XA transaction 94
inbound adapter classes 37
  Connection pooling 37
  Event polling 38
  inbound callback event 38
  properties 37, 38
inbound JCA adapter classes 50
installation 11
Interaction 49
InteractionSpec 49
Interface (CCI), Common Client 4

J

J2C Resource Adapter Properties window 33
Java 2 Connector Architecture JCA 1.5 specification 10
Java Connector Project 9
JavaBean properties 66
Javadoc 10
javax.resource.spi.ActivationSpec 50
JCA 1.5 specification, Java 2 Connector Architecture 10
JCA adapter classes, inbound 50
JCA adapter classes, outbound 49
JCA enterprise metadata discovery classes 50

K

KiteString sample 12

L

LocalTransaction 49
log messages 178
Logging
  Common Event Infrastructure 184
  events 184

M

ManagedConnectionFactory 50, 67
ManagedConnectionMetaData 50
MangedConnection 50
Message Listener Property 67
Metadata
  API 119
  application-specific information 7
  artifact types 122
  data bindings 122
  factory classes 120
  generated records 122
  generic records 122
  implementation 122
  interfaces 120
  SDOFactory 120
  supported runtimes 122
  TypeFactory 120
metadata discovery classes, enterprise 39, 47
metadata discovery classes, JCA enterprise 50
metadata discovery, enterprise 3
Monitoring
  adapters 181, 185
  Application response measurement (ARM) 187
  Common Event Infrastructure 181, 184
  Common Event Infrastructure (CEI) 181
  element schema 183
  Event point 182
  Event source 182
  EventSourceContext 181
  InboundEventDelivery 185
  InboundEventRetrieval 185
  interface 181
  logging 184
  Outbound 185
  Performance monitoring infrastructure (PMI) 185, 186, 187
  types of 181
  Unique Id 183
  Xsd schema 184

N

namespace definition 136
New Connector Project Wizard 9

O

one way callback events 90
operating system requirements 11
Operations
  inbound 72
  outbound 73
  standard 72, 73
  top-level 72
Outbound
  operations 73
  standard 73
outbound adapter classes 40
  Command pattern 37
  Local transaction support 36
outbound adapter classes *(continued)*
  properties 35, 36, 37
  WebSphere Resource Adapter 35
  XA transaction support 36
outbound JCA adapter classes 49

P

pattern classes, command 43
Performance monitoring infrastructure (PMI) 185
plug-in, Eclipse 9
Project
  New Connector Project wizard 29
Project facets 32
project, Enterprise Application Archive (EAR) 31
Project, Java Connector 9

R

Request and response callback events 90
resource adapter archive (RAR) 2
Resource Adapter Deployment Descriptor editor 57, 68
Resource Adapter Deployment Descriptor Editor 9
resource adapter properties 33
Retrieve 77
RetrieveAll 77

S

sample, KiteString 12
sample, TwineBall 12
samples 9
service data object (SDO) 70
Service-Oriented Architecture (SOA) 70
software requirements 12
specification, Java 2 Connector Architecture JCA 1.5 10
Standard operations 72
Structured record implementation 163
  Clone 165
  Close 165
  Extract 165
  getNext() 164
  initializeInput 163
  initializeOutput 163
  methods 163, 164, 165
  sample code 163, 164, 165
  setManagedConnection 164

T

Toolkit, WebSphere Adapter 1
trace levels 177
trace messages 176
trace messages, hiding confidential data 176
TwineBall sample 12

U

Update, after-image 75

V

verbs
    usage
        business graph 71

W

WBIActivationSpec 37
WBIAdapterTypeImpl 39, 141
WBIConnection 35, 102
WBIConnectionFactory 35, 101
WBIConnectionRequestInfo 104
WBIDataBindingGenerator 38
WBIDataDescriptionImpl 39, 151
WBIInboundConnectionConfigurationImpl 39, 146
WBIInboundConnectionTypeImpl 39, 143
WBIInboundServiceDescriptionImpl 39, 155
WBIInteraction 35, 102
WBIInteractionSpec 35, 103
WBILocalTransaction 35
WBIManagedConnection 35, 100
WBIManagedConnectionFactory 35, 99
WBIMetadataDiscoveryImpl 39, 139
WBIMetadataEditImpl 39, 151
WBIMetadataImportConfigurationImpl 39, 150
WBIMetadataObjectImpl 39, 149
WBIMetadataSelectionImpl 39, 149
WBIMetadataTreeImpl 39, 147
WBIOutboundConnectionConfiguration 39
WBIOutboundConnectionConfigurationImpl 144
WBIOutboundConnectionTypeImpl 39, 143
WBIOutboundServiceDescriptionImpl 156
WebSphere Adapter 3
WebSphere Adapter Toolkit 1
WebSphere Adapter Toolkit tasks 11
WebSphere Integration Developer 29

X

XML Schema Definition 68

Printed in USA